

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

腾讯公司资深研发工程师多年后台开发经验总结，获腾讯、Facebook、微软、阿里、百度多位资深技术专家高度认可。

完整勾勒后台开发技术能力体系，多维度讲解了成为一名后台开发工程师所需掌握的核心技术、开发工具和实践方法，后台工程师修炼必读！



徐晓鑫 著

Server-side Development: Technology and Practices

后台开发 核心技术与应用实践



机械工业出版社
China Machine Press

内 容 简 介

因为后台开发所需要的技术广泛而坚深，要成为一名后台开发工程师门槛很高，所以相关人才比较紧缺。作者是在腾讯工作多年的后台开发工程师，不仅技术精湛，而且在处理大量实际业务的过程中积累了丰富的开发经验。在这本书中，她不仅首次为后台开发工程师勾勒出了完整的知识能力体系结构图，而且还对后台开发工程师所需要掌握的大量复杂的技术知识进行了提炼、剥离和整合，专注于成为一名后台开发工程师所需掌握的核心技术、开发工具和实践方法，大幅度降低后台开发工程师的学习曲线。本书的内容获得了来自腾讯、Facebook、微软、阿里、百度的多位资深技术专家的高度认可。

全书一共13章，在逻辑上分为六大部分：

第一部分（第1~3章）介绍了编程语言方面的知识，包括常用语法、类与常用STL的使用。

第二部分（第4~5章）介绍了编译原理和调试方法相关的知识，编译原理包括编译与链接的具体过程、Makefile的编写、目标文件的内容与处理目标文件相关工具的使用，调试方法主要介绍了strace、gdb、top、ps与valgrind工具的使用等。

第三部分（第6~8章）介绍了网络相关的知识，包括TCP协议的关键知识点和TCP server的实现，网络IO模型和select、poll与epoll三个重要函数的使用，还有ping、tcpdump、netstat和lsof这四个网络分析工具的使用。

第四部分（第9~11章）主要是多线程、进程和进程间通信相关的知识，包括多线程的使用、多线程的同步和重入问题，进程方面有父子进程、僵尸进程、守护进程和进程间通信的方式。

第五部分（第12章）主要是HTTP协议的介绍与使用、CGI的设计原理、实现和FASTCGI的简单介绍。

第六部分（第13章）通过常用类库JsonCPP和Protobuf的使用，演示如何使用第三方库。



Server-side Development: Technology and Practices

后台开发 核心技术与应用实践

徐晓鑫 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

后台开发: 核心技术与应用实践 / 徐晓鑫著. —北京: 机械工业出版社, 2016.8 (2016.12 重印)

ISBN 978-7-111-54339-8

I. 后… II. 徐… III. 网络-开发 IV. TP393.092

中国版本图书馆 CIP 数据核字 (2016) 第 167884 号

后台开发: 核心技术与应用实践

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 李 艺

责任校对: 董纪丽

印 刷: 三河市宏图印务有限公司

版 次: 2016 年 12 月第 1 版第 2 次印刷

开 本: 186mm×240mm 1/16

印 张: 26.5

书 号: ISBN 978-7-111-54339-8

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Preface 序

C++ 可能是计算机历史上最早被发明的高级程序语言，同时也是当今最活跃的程序设计语言之一。C++ 很强大，强大到你可以使用它做任何层面的开发；C++ 也很脆弱，脆弱到需要程序员自己去控制内存回收，一个不小心就会使整个程序 Core Dump。C++ 语言的创始人 Bjarne Stroustrup 曾私下承认，为了提高 C++ 程序员的薪水和地位，在设计 C++ 编译器版本过程中有意地增加了 C++ 语言的难度，使 C++ 更偏向于资深程序员的使用习惯，提高学习门槛，从而增加 C++ 程序员的身价。学习曲线的增加并不是没有任何回报的，在服务端后台开发、处理多并发的海量网络请求方面，C++ 语言有天然的优势。因此，当应用的用户量、并发量迅速增长，达到一定量级之后，后端服务的技术架构都会转变为 Linux C++。

要做一名优秀的使用 C++ 进行后台开发的程序员，只掌握 C++ 语言是远远不够的，还需要掌握如何进行编译、链接、调试，如何使用网络协议、IO 模型和一些常用的类库，等等。我曾经面试过不少后台开发程序员，他们往往很重视语言本身，但是对一些语言之外的东西理解不够透彻，影响了他们的技术发展。我也读过不少相关方面的技术书籍，往往都过多地停留在语言层面，忽略了实际开发工作中需要用到的知识。

晓鑫在腾讯从事开发工作多年，有丰富的后台开发经验，她从实际的后台开发经验出发，讲解了后台开发中需要用到的方方面面的知识。从 C++ 语言出发，又不止于 C++ 语言，本书可以说是一本 Linux C++ 后台开发的实战典范。当知道晓鑫在写这么一本书的时候，我真心想为国内的众多开发者感到高兴。如果读者有意愿成为一名从事 Linux 后台开发的程序员，本书无疑是一本最佳的参考书籍。

研发是一项讲究实战的工作，一切不从实际工作出发的技术书籍都是纸上谈兵，没有实际意义。一本优秀的技术书籍应该是这样的：当读者按照书中的内容进行实操的时候，读者写的每一行代码都是有价值的，能够在实际工作中派上用场。本书恰好做到了这一点。这是

一位技术书籍作者对读者的起码诚意。

软件工程师是一种需要坚定、踏实、精益求精的“工匠精神”的职业，心浮气躁、得过且过的态度不可能把代码写好。老一辈的人说“字如其人”，在软件领域，我们同样可以说“代码如其人”，一个人的行事风格和为人态度都会体现到他所写的代码上面。按照晓鑫的书去学习，读者可以潜移默化地学习到她多年后台开发所练就的“工匠精神”。我想，相对于所学习到的知识，这于一个工程师来说更为重要。

黄世飞

腾讯云平台技术总监

0.1 什么是后台开发

听到“后台开发”这个词，估计读者心中都或多或少会有一些自己的感性认识，这种认识可能有一些差别，但估计大部分人都有这么一种看法：“后台开发”是编写一些用户看不见的程序，也就是非界面的程序，既不是网页，也不是 App，更不是桌面程序，因为这些都是用户看得见的（被称为“前台开发”）。这种感性认识在一定程度上是正确的，但是它不够具体，也不够全面。

我们这里所说的“后台开发”的确是用户“看不见”的部分，但是还有很多界面性的程序是给企业内部人员使用的，这些虽然是界面程序，但是对于最终用户来说也是“看不见”的。举个例子，开发一个电子商务网站，提供给客户进行商品购买的网页是用户看得见的，不属于“后台”，但是电商网站内部员工使用的“用户管理系统”，“订单管理系统”等，也是用户看不见的，但它们不属于本书中所指的“后台”。在某些场合，或者某些人的习惯中，这些内部使用的系统也叫“后台”，这几种说法都没有错，希望读者在听到的时候，知道说话人具体指的是什么。

本书介绍的“后台开发”指的是“服务端的网络程序开发”，从功能上可以具体描述为：服务器收到客户端发来的请求数据，解析请求数据后处理，最后返回结果，如图 0-1 所示。

这里的 SERVER 就是本书所指的“后台程序”，或者“服务器”。SERVER 接收请求的方式既可以通过 TCP 请求包，也可以是 HTTP 请求包（其实也是 TCP 连接）。如果是 TCP 请

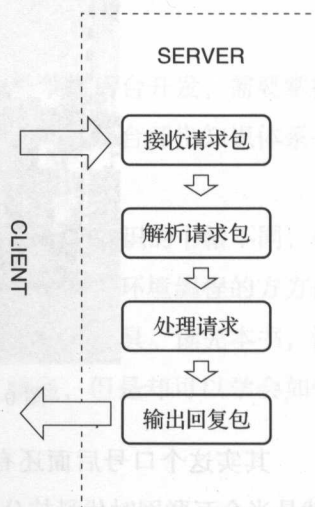


图 0-1 后台开发的步骤

求，二进制的格式会常见一些；如果是 HTTP 方式的请求，请求包的格式一般是 JSON 或者 XML，或者自定义的 ASCII 文本。解析请求包的方式自然是与请求包的格式相对应的，接收到的是什么格式的包，就用对应的格式解析（如果是自定义的格式，就按照自定义的方式去解析）。“处理请求”这一步是后台程序的具体业务逻辑的体现。很多封装好的后台框架会把其他三步都做好，但是这一步还是需要开发者自己去实现，因为只有开发者自己清楚，程序是要去做“登录”还是去做“注册”的事情。“输出回复包”和“接收请求包”是对应的，一般来说，收到的是 JSON，那么回复的也是 JSON，收到的是 XML，那么发送的也是 XML，其他格式也是一样的。这四个步骤是所有后台程序都会有的，无论使用什么语言去实现，都可以看到这四个步骤的影子。

CLIENT 指的是向 SERVER 发起请求，并接收 SERVER 回复的一方，通常称为“客户端”。既然后台程序是通过 TCP 或者 HTTP 接收和回复消息的，那么只要是能够发起 TCP 或者 HTTP 连接的都可以作为客户端，可以是浏览器、PC 端的程序、安卓应用、IOS 应用，等等。

0.2 时间就是金钱，效率就是生命

上世纪 80 年代，在改革开放初期的深圳蛇口，为了加快蛇口港的建设，一块“时间就是金钱，效率就是生命”的巨幅标语矗立在蛇口工业区的马路边（如图 0-2 所示），拉开了特区建设的序幕。



图 0-2 1981 年矗立在深圳蛇口工业区的巨幅标语

其实这个口号后面还有两句：“安全就是法律，顾客就是皇帝”，这四句加一起，简直就是当今互联网时代科技公司安身立命的根本。互联网最讲究效率并且一切都从用户体验出

发。在腾讯、百度和阿里这样的互联网公司里，每一次的版本发布都是和时间在赛跑。各种以效率为核心的开发团队合作模式被创造：敏捷开发、极限编程、SCRUM、结对编程，双周迭代，等等。

写下这些文字的时候是我在腾讯工作的第五个年头，这五年让我对效率有了更深刻的认识。还是一个学生的时候，和大家一样，我也曾一字不落地读过《UNIX 环境高级编程》，《UNIX 环境网络编程》一二三卷，《TCP/IP 详解》一二三卷，《C++ Primer》等书籍，这些都是非常经典的开发书籍。它们的共同特点是大而全，不漏掉任何一个知识点，并且每个知识点都讲得非常详细。但在实际的开发工作中，可能用到的知识点只有 20%，其他的 80% 则很少用到。这也是我写这本书的初衷：用最短的篇幅，讲解实际后台开发中用到的核心知识点，让读者可以快速进入到实际的开发工作中。

也许有读者会觉得这很急功近利，不利于组建完整的知识体系。其实，软件开发是一门讲究实操的技术，知道多少并不重要，重要的是能够用好多少。如果把一本经典书籍读 3 遍，但是没有写过一行代码，那可以认为是没有读。边写代码边读书才是最好的学习方式，在读一本技术书籍的时候，最好让自己快速进入写代码的状态，一边写代码，一边通读书籍，在具体需要用到书上某个技术点的时候，再回头仔细阅读相关的章节。在这个循环往复的过程中，才能把书上的知识点转化为自己的知识点。完成多个这样的循环后，再回过头来审视自己已经掌握的知识点，把一些没有掌握的知识点搞清楚。这样的学习过程实际上更有利于完善自己的知识体系。

0.3 后台开发的知识体系

接下来简单介绍一下后台开发的知识体系，也就是说，要完整掌握后台开发，需要掌握哪些知识点，这些知识点也是本书会讲解到的知识点。图 0-3 展示了对后台开发知识体系一个比较全面的梳理。

以上这六部分知识点会是本书将会覆盖的内容。与专门介绍某一类知识的书籍不同，比如《C++ Primer》介绍 C++ 的方方面面，《UNIX 环境编程》介绍 UNIX 环境编程的方方面面，本书从“实战”的角度出发，介绍“后台开发”需要用到的知识和工具。读完本书，读者可能不会对 C++ 精通，不会对 Linux 精通，也不会对 TCP/IP 精通，但是却可以学会如何进行“后台开发”，这些“精通”可以一个个慢慢补全。

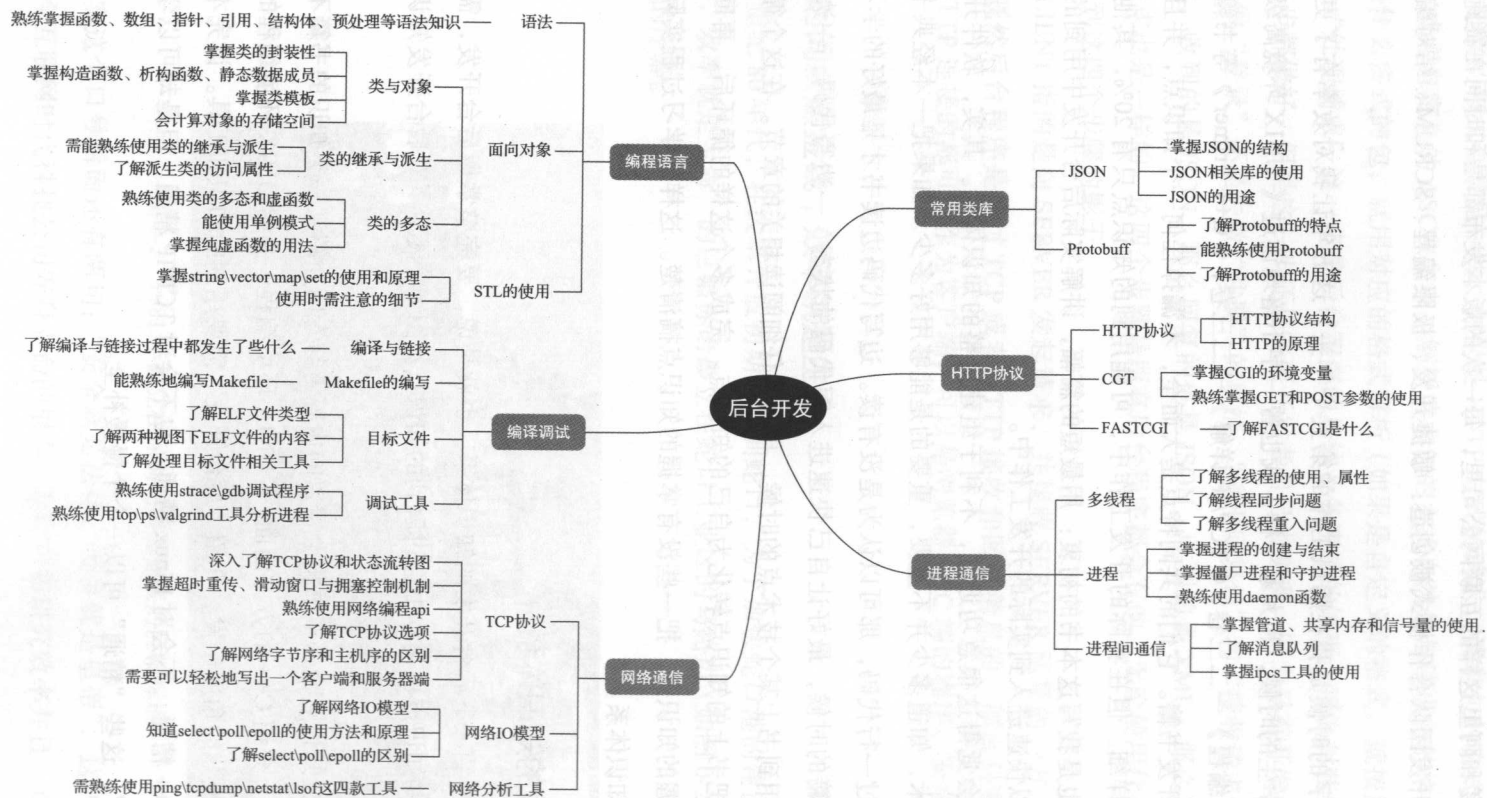


图 0-3 后台开发工程师技术能力体系图

细心的读者可能会发现，编程语言和编辑器只是其中的一小部分，要把后台开发做好，需要掌握的东西比想象中的要多。同时，这些知识点既有一些纯理解性的内容，也有一些工具性的东西。这也是程序开发的最大特点，既要掌握理论的东西，也要掌握相应的工具，这样才能把理论的知识用起来。在实际开发中，这些工具性的知识可能更重要，因为理论是脑子里想的，但工具产生的东西才是真正的产出，才是开发者最终需要的。

0.4 如何阅读本书

本书以 C++ 为编程语言，讲述后台开发的核心技术与应用实践。全书共 13 章，在逻辑上分为以下六部分：

第一部分为第 1 ~ 3 章，主要是编程语言方面的知识，包括函数、函数重载、函数模板、数组、指针、引用、结构体和预处理的使用；面向对象的介绍，包括类的使用、继承与派生和类的多态；常用 STL 的介绍，包括 string、vector、map 和 set 的使用方法与原理。如果读者已经对 C++ 非常了解，可以跳过这部分，也可以配合《C++ Primer》一起阅读。

第二部分为第 4 ~ 5 章，主要是编译原理和调试方法相关的知识。编译原理相关知识包含编译与链接的具体过程，makefile 的编写、目标文件的内容与处理目标文件相关工具的使用；调试方法相关内容主要介绍了用 strace 分析系统调用、用 gdb 调试进程与分析 core dump 文件、用 top 命令分析系统负载情况、用 ps 命令查看系统进程和用 valgrind 工具分析进程的内存使用情况等。

第三部分为第 6 ~ 8 章，主要是网络相关的知识，包括 TCP 协议的关键知识点和 TCP server 的实现，网络 IO 模型和 select、poll 与 epoll 三个重要函数的使用，还有 ping、tcpdump、netstat 和 lsof 这四个网络分析工具的使用。掌握这部分知识，读者可以自己独立实现能处理海量请求的 TCP server。

第四部分为第 9 ~ 11 章，主要是多线程、进程和进程间通信相关的知识，包括多线程的使用、多线程的同步和重入问题，父子进程、僵尸进程、守护进程和进程间通信的方式。读者可以配合《UNIX 环境高级编程》一起阅读。

第五部分是第 12 章，主要是 HTTP 协议的介绍与使用、CGI 的设计原理与实现和 FASTCGI 的简单介绍。掌握这部分知识，读者可以轻松实现 Web 应用的后台交互部分。

第六部分是第 13 章，通过常用类库 JsonCPP 和 Protobuf 的使用，演示如何使用第三方库。

如果读者是后台开发的新手，建议从第 1 章开始阅读，如果读者已经有后台开发的经验，可以直接选择感兴趣的章节阅读。

0.5 勘误和资源

由于水平有限，加之编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。为了更好地与读者交流，我专门创建了一个微信公众号，读者可以通过以下二维码关注，与我进行交流。



书中的全部源代码可以从华章网站 (www.hzbook.com) 下载。

0.6 致谢

首先要感谢腾讯公司，让我可以在后台开发的领域里驰骋。

其次要感谢机械工业出版社华章公司的杨福川、高婧雅和李艺，感谢你们在我写作过程中提供的支持，因为有了你们的鼓励和帮助，我才能顺利完成全部书稿。

谨以此书献给我亲爱的家人，以及热爱软件开发的朋友们！

Contents 目 录

序 绪论

第1章 C++ 编程常用技术 1

1.1 第一个 C++ 程序 1

1.2 函数 3

1.3 数组 6

1.4 指针 8

1.5 引用 12

1.6 结构体、公用体、枚举 14

1.6.1 结构体、公用体、枚举的概念 14

1.6.2 结构体、公用体在内存单元 占用字节数的计算 18

1.7 预处理 20

1.8 本章小结 25

第2章 面向对象的 C++ 26

2.1 类与对象 26

2.2 继承与派生 49

2.3 类的多态 57

2.4 本章小结 64

第3章 常用 STL 的使用 65

3.1 STL 是什么 65

3.2 string 66

3.3 vector 77

3.3.1 vector 是什么 77

3.3.2 vector 的查增删 78

3.3.3 vector 的内存管理与效率 86

3.3.4 Vector 类的简单实现 90

3.4 map 96

3.4.1 map 是什么 96

3.4.2 map 的查增删 96

3.4.3 map 的原理 109

3.5 set 111

3.5.1 set 是什么 111

3.5.2 set 的查增删 112

3.6 本章小结 116

第4章 编译 117

4.1 编译与链接 117

4.2 makefile 的撰写 131

4.3 目标文件 135

4.3.1 ELF 的文件类型	135	6.1.5 TCP 滑动窗口	200
4.3.2 链接视图下的 ELF 内容	136	6.1.6 TCP 拥塞控制	202
4.3.3 执行视图下的 ELF 内容	142	6.2 TCP 网络编程 API	205
4.3.4 阅读 ELF 文件的 工具——readelf	144	6.3 实现一个 TCP server	211
4.3.5 获得二进制文件里 符号的工具——nm	144	6.4 TCP 协议选项	215
4.3.6 减少目标文件大小 的工具——strip	146	6.5 网络字节序与主机序	233
4.4 本章小结	147	6.6 封包和解包	233
第 5 章 调试	148	6.7 本章小结	247
5.1 strace	148	第 7 章 网络 IO 模型	248
5.2 gdb	156	7.1 4 种网络 IO 模型	248
5.3 top	164	7.2 select	256
5.4 ps	165	7.3 poll	267
5.5 Valgrind	168	7.4 epoll	277
5.5.1 Valgrind 概述	168	7.5 本章小结	289
5.5.2 Linux 程序内存空间布局	170	第 8 章 网络分析工具	290
5.5.3 内存检查原理	175	8.1 ping	290
5.5.4 Valgrind 安装	176	8.2 tcpdump	292
5.5.5 Valgrind 使用	177	8.3 netstat	294
5.6 本章小结	187	8.4 lsof	296
第 6 章 TCP 协议	188	8.5 本章小结	298
6.1 TCP 协议	188	第 9 章 多线程	299
6.1.1 网络模型	188	9.1 多线程是什么	300
6.1.2 TCP 头部	191	9.2 多线程的创建与结束	301
6.1.3 TCP 状态流转	193	9.3 线程的属性	307
6.1.4 TCP 超时重传	196	9.4 多线程同步	312
		9.5 多线程重入	332
		9.6 本章小结	333

第 10 章 进程	334	11.6 本章小结	374
10.1 程序与进程	334	第 12 章 HTTP 协议	375
10.2 进程的创建与结束	335	12.1 HTTP 协议工作流程	375
10.3 僵尸进程	342	12.2 HTTP 协议结构	376
10.4 守护进程	347	12.3 HTTPS	383
10.5 本章小结	351	12.4 CGI	386
第 11 章 进程间通信	352	12.5 FastCGI	397
11.1 管道	352	12.6 本章小结	398
11.2 消息队列	358	第 13 章 常用类库	399
11.3 共享内存	362	13.1 JSON	400
11.4 信号量	368	13.2 Protobuf	405
11.5 ipc 命令	373	13.3 本章小结	409

我们通过固定格式和固定格式的语言来描述数据，以便人为我们做管理。语言有很多种，包括汉语、英语、日语、韩语等，虽然它们的词、句格式都不一样，但是可以达到同样的目的，我们可以在计算机上通过语言来描述数据。同样，我们也可以通过“语言”来描述计算机，让计算机为我们做管理。这样的语言就叫做编程语言。

C语言是1972年由美国贝尔实验室的D.K.Ritchie设计发明的。它作为计算机上的人机接口语言，大多数程序员都会使用。C语言是面向过程的编程语言，随着软件规模的增大，用C语言编程变得越来越复杂。C++是由美国贝尔实验室的Bjarne Stroustrup博士及团队于20世纪80年代设计出的语言，它是在C语言的基础上，保留了C语言所有的优点，与C语言兼容，并且增加了面向对象的机制。程序员写的程序基本上可以不加修改地用于C++开发工具。从C++的名字可以看出来，它结合了C++和C++的面向过程的结构化程序设计，也可用于面向对象的程序设计，是一种功能强大的混合型程序设计语言。

本章主要讲述C++中的常用技术，让读者能够快速上手。由于篇幅有限，部分内容将不再赘述。

1.1 第一个C++程序

刚开始接触一门编程语言，一般会从一个输出Hello world的程序开始。

【例 1.1】用程序输出Hello world

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world" << endl;
    return 0;
}
```



第1章

Chapter 1

C++ 编程常用技术

我们通过固定格式和固定词汇的“语言”来影响他人，让他人为我们做事情。语言有很多种，包括汉语、英语、法语、韩语等，虽然它们的词汇和格式都不一样，但是可以达到同样的目的，我们可以选择任意一种语言去与他人交流。同样，我们也可以通过“语言”来影响计算机，让计算机为我们做事情，这样的语言就叫作编程语言。

C 语言是 1972 年由美国贝尔实验室的 D.M.Ritchie 设计成功的，它是为计算机专业人员设计的，大多数系统软件和许多应用软件都是用 C 语言编写的。但是随着软件规模的增大，用 C 语言编写程序渐渐显得有些吃力了。C++ 也是由美国贝尔实验室的 Bjarne Stroustrup 博士及其同事于 20 世纪 80 年代初在 C 语言的基础上开发成功的。C++ 保留了 C 语言原有的所有优点，与 C 语言兼容，并且增加了面向对象的机制。用 C 语言写的程序基本上可以不加修改地用于 C++ 开发工具。从 C++ 的名字可以看出它是 C 的超集。C++ 既可用于面向过程的结构化程序设计，又可用于面向对象的程序设计，是一种功能强大的混合型程序设计语言。

本章主要讲述 C++ 中的常用技术，让读者可迅速地、由浅入深地熟悉这门语言。

1.1 第一个 C++ 程序

刚开始接触一门编程语言，一般会从写一个输出 Hello world 的程序开始。

【例 1.1】用程序输出 Hello world。

```
#include<iostream>
using namespace std;
int main()
{
```



```

    cout<<"Hello world."<<endl;
    return 0;
}

```

把上述程序编写在一个叫 `helloworld.cpp` 的文件中，并将它放到 Linux 机器上的某个目录下，执行 `g++ helloworld.cpp` 命令，会在该目录下生成 `a.out` 文件。执行 `./a.out` 命令，即可得到输出结果：Hello world。

先看程序的第一行 (`#include<iostream>`)，这不是一个 C++ 语句，是一个预处理语句，编译器的预处理器把输入输出流的标准头文件包括在本程序中，所以不需要在句末加分号 (`;`)。include 一个文件，就是把这个文件的所有内容都加进来。图 1-1 展示了包含文件的过程。

如图 1-1 所示，include 一个 .h 文件，就是等于把整个 .h 文件给复制到程序中，include 一个 .cpp 文件也是如此。

除了 `#include<>` 的方式来包含一个头文件，还会见到 `#include"` 的方式来包含一个头文件。而 `#include<>` 与 `#include"` 的区别是：`#include<>` 常用来包含系统提供的头文件，编译器会到保存系统标准头文件的位置查找头文件；而 `#include"` 常用于包括程序员自己编号的头文件，用这种格式时，编译器先查找当前目录是否有指定名称的头文件，然后从标准头目录中进行查找。

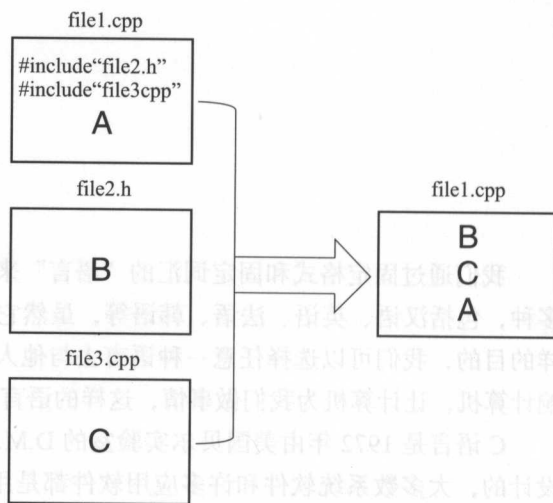


图 1-1 include 文件的原理

还经常会看到 `#include<iostream>` 和 `#include<iostream.h>` 的使用。事实上，`#include<iostream>` 和 `#include<iostream.h>` 是不一样的，因为 `iostream` 和 `iostream.h` 是两个不同的文件，前者没有后缀。实际上，在你的编译器 include 文件夹里面可以看到，两个文件打开后，里面的代码是不一样的。后缀为 .h 的头文件在 C++ 标准已经明确提出不再支持了，早些的 C 语言为了实现将标准库功能定义在全局空间里，声明放在在带 .h 后缀的头文件里。C++ 标准为了和 C 语言区别开，也为了正确使用命名空间，规定头文件不再使用后缀 .h。因此，当使用 `<iostream.h>` 时，相当于在 C 中调用库函数，使用的是全局命名空间，也就是早期的 C++ 实现方法。换句话说，`iostream` 是 `iostream.h` 的升级版，大部分的头文件都有一个不带 .h 扩展名的文件与之相对应。不过有个特例，`<string>` 并非 `<string.h>` 的升级版。

再看程序的第二行：“`using namespace std;`”中使用了命名空间 `std`。命名空间是为了让大量类共存而不至于引起冲突而设计的。C++ 标准函数库的所有元素都被声明在一个命名空间中，这就是 `std` 命名空间。为了能够访问它的功能，使用这条语句来表达将使用标准命名空间中

定义的元素。这条语句在使用标准函数库的 C++ 程序中频繁出现，本书中大部分例子的代码中也将用到它，需要注意的是，最好不要在头文件中使用命名空间，否则容易造成命名冲突。

继续看程序的第三行：“int main()”，这是主函数 (main function) 的起始声明。主函数是所有 C++ 程序的运行的起始点。不管它是在代码的开头、结尾还是中间，此函数中的代码总是在程序开始运行时第一个被执行。main 后面跟了一对圆括号 ()，表示它是一个函数。C++ 中所有函数都跟有一对圆括号 ()，括号中可以有一些输入参数。如例 1.1 中显示，主函数 (main function) 的内容紧跟在它的声明之后，由花括号 {} 括起来。

程序的第四行：“cout<<"Hello world."<<endl;”是本程序中最重要。cout 是 C++ 中的标准输出流（通常为控制台，即屏幕），这句话把一串字符串（本例中为 Hello World）插入到输出流中。cout 的声明在头文件 iostream 中，所以要想使用 cout 必须将该头文件包括在程序开始处。注意这个句子以分号结尾。分号标示了一个语句的结束，C++ 的每一个语句都必须以分号结尾。C++ 程序员最常犯的错误之一就是忘记在语句末尾写上分号。

最后一行 (return 0;) 中返回语句 (return) 标志主函数 main() 执行结束，并将该语句后面所跟代码（在本例中为 0）返回。这是在程序执行没有出现任何错误的情况下最常见的程序结束方式。在后面的例子中会看到所有 C++ 程序都以类似的语句结束。

1.2 函数

1. 函数的定义

一个 C 程序是由若干个函数组成的，C 语言被认为是面向函数的语言，而 C++ 面向过程的程序设计也沿用了 C 语言使用函数的方法。在 C++ 面向对象的程序设计中，主函数以外的函数大多是被封装在类中的。主函数或其他函数可以通过类对象调用类中的函数。无论是 C 还是 C++，程序中的各项操作基本上都是由函数来实现的，程序编写者要根据需要编写一个个函数，每个函数用来实现某一功能。因此，读者必须掌握函数的概念以及学会使用和设计函数。

定义函数的一般格式是：

```
返回值类型 函数名 ([形参])
{
    函数体
}
```

在定义函数时函数名后面括号中的变量名称是形参。在主调中调用一个函数时，函数名后面括号中的参数是实参。

【例 1.2】 函数、形参、实参的使用举例。

```
#include<iostream>
using namespace std;
```

```
int min(int a,int b){ //这里的min就是函数名,a、b是形参,
                      //返回值是一个int 整型
    if(a<b)return a;
    else return b;
}
int main(){
    int a=10,b=1;
    cout<<min(a,b)<<endl; //这里的a、b是实参
    return 0;
}
```

函数的执行结果是:

1

例 1.2 中定义了一个函数,有 2 个整型的参数,且返回值是整型的,在 main 函数中调用 min 函数时,传入的是实参。

形参与实参的区别是:形参只有被调用时才分配内存单元,在调用结束时,立即释放所分配的内存单元。实参与形参的类型应相同或赋值兼容。

2. 函数重载

C++ 允许用同一函数名定义多个函数,但这些函数必须参数个数不同或类型不同,这就是函数重载。例 1.3 说明了函数重载的使用方法。

【例 1.3】求不同类型的数中的最小者。

```
#include<iostream>
using namespace std;
int min(int a, int b, int c){
    if(a>b)a=b;
    if(a>c)a=c;
    return a;
}
long long min(long long a,long long b, long long c){
    //与上面那个函数的差别是参数的类型不同
    if(a>b)a=b;
    if(a>c)a=c;
    return a;
}
double min(double a, double b){
    //这个函数与以上的差别是只有 2 个参数
    if(a-b>(1e-5))a=b;
    return a;
}
int main(){
    int a=1,b=2,c=3;
    cout<<min(a,b,c)<<endl;
    long long a1=100,b1=200,c1=300;
    cout<<min(a1,b1,c1)<<endl;
    double a2=1.1,b2=2.2;
    cout<<min(a2,b2)<<endl;
```

```

    return 0;
}

```

程序的执行结果是:

```

1
100
1.1

```

这里分别需要比较 3 个整数、3 个长整数和 2 个浮点数, 并获得各组中的最小值。例 1.3 中分别定义了 3 个函数, 而且函数名都是一样的, 不过参数个数不一样或者参数类型不一样, 这就是使用了函数重载来实现功能。

在使用函数重载时, 同名函数的功能应当相同或相近, 不要用同一函数名去实现几个完全不相干的功能, 这样虽然程序能运行, 但是可读性不好, 会让人觉得莫名其妙。

3. 函数模板

函数模板, 实际上是建立一个通用函数, 其函数类型和形参不具体指定, 而用一个虚拟的类型来代表, 这个通用函数就是函数模板。凡是函数体相同的函数都可以用这个模板来代替, 而不用定义多个函数, 实际使用时只需在模板中定义一次就可以了。在调用函数时, 系统会根据实参的类型来取代模板中的虚拟类型, 从而实现不同函数的功能。

定义函数模板的一般格式是:

```
template<typename T>
```

下面的程序说明了函数模板的使用方法。

【例 1.4】函数模板使用举例。

```

#include<iostream>
using namespace std;
template<typename T>
T min(T a,T b,T c){
    if(a>b)a=b;
    if(a>c)a=c;
    return a;
}
int main(){
    int a=1,b=2,c=3;
    cout<<min(a,b,c)<<endl;
    long long a1=1000000000,b1=2000000000,c1=3000000000;
    cout<<min(a1,b1,c1)<<endl;
    return 0;
}

```

程序输出结果:

```

1
1000000000

```

例 1.4 中定义了一个函数模板，用来获得 3 个数中的最小者。若传入 3 个整型的，函数就将虚拟类型 T 转成 int 去执行；若传入 3 个长整型的，函数就将 T 转化成 long long 去执行。这样就可以不用定义类型不同的函数了，只需一个函数模板即可搞定。

在编写函数模板时，可以先写一个函数，然后把其中的变量类型都替换成虚拟类型即可。可以看到，用函数模板比函数重载更方便，但是它只适用于函数个数相同而类型不同的情况。

1.3 数组

1. 数组的定义

数组是相同类型数据的集合。引入数组就不需要在程序中定义大量的变量，大大减少程序中变量的数量，使程序精炼，而且数组含义清楚，使用方便，明确地反映了数据间的联系。许多好的算法都与数组有关。熟练地利用数组，可以大大地提高编程的效率，加强程序的可读性。例 1.5 展示了数组的使用方法。

【例 1.5】 一维数组与二维数组使用举例。

```
#include<iostream>
using namespace std;
int main(){
    int a[10]={1,2,3,4,5,6,7,8,9,10};
    // 数组 a, 类型为 int 整型, 有 10 个元素, 是一个一维数组

    int i,j;
    for(i=0;i<10;i++){
        cout<<a[i]<<" ";
        // 从 a[0]~a[9]
    }
    cout<<endl;
    int b[2][2]={1,2,3,4};
    // 数组 b, 类型为 int, 有 4 元素, 是一个二维数组
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            cout<<b[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

程序的执行结果是：

```
1 2 3 4 5 6 7 8 9
1 2
3 4
```

例 1.5 中数组 a 是一个一维数组，有一个下标；数组 b 是一个二维数组，有两个下标。

一个数组在内存中占用的存储单元是连续的，也就是说一个数组在内存中占用一片连续的存储单元。在 32 位的机器上，一个 int 类型的值占 4Byte，如果 a[0] 的地址是 2000，那么

a[1] 的地址就是 2004, a[2] 的地址就是 2008, a[3] 的地址就是 2012……如此类推。这里的“地址”，大家可以简单理解为在内存中的一个标识，详细内容会在本章的 1.4 节中进行介绍。

2. 字符数组

字符数组，就是一个用来存放字符数据的数组，示例如下：

```
char str[10]="Book";
```

其中，str 就是一个字符数组，并且 str[0]='B', str[1]='o', str[2]='o', str[3]='k'。C++ 中用 '\0' 来标识一个字符串的结束，这里，str[4]~str[9] 都是 '\0'。通常用 strlen() 函数来计算一个字符串的长度，故 strlen(str) 的值是 4。与 strlen() 函数比较容易混淆的是 sizeof() 函数，这里 sizeof(str) 的值是 10。

strlen 与 sizeof 的区别如下所示：

(1) strlen() 是函数，在运行时才能计算。参数必须是字符型指针 (char*)，且必须是以 '\0' 结尾的。当数组名作为参数传入时，实际上数组已经退化为指针了。它的功能是返回字符串的长度。

(2) sizeof 是运算符，而不是一个函数，在编译时就计算好了，用于计算数据空间的字节数。因此，sizeof 不能用来返回动态分配的内存空间的大小。sizeof 常用于返回类型和静态分配的对象、结构或数组所占的空间，返回值跟对象、结构、数组所存储的内容没有关系。

当参数分别如下时，sizeof 返回的值表示的含义如下所述。

1) 数组——编译时分配的数组空间大小，如：

```
char a[10]="hello";
```

因为 char 占 1Byte，所以 sizeof(a) 的值是 10*1=10Byte。

2) 指针——存储该指针所用的空间大小，如：

```
char *str="I am from China."
```

因为 str 存储的是一个字符指针，所以 sizeof(str) 是指针所占的空间，即是 4Byte。

3) 类型——该类型所占的空间大小，如：

```
int b=10;
```

因为在 32 位的机器上，int 类型占 4Byte，所以 sizeof(b) 的值是 4Byte。

4) 对象——对象的实际占用空间大小，如：

```
class Class_Sample{
    int a,b;
    int func();
}Class_a;
```

两个 int 类型的值是 8Byte，所以 sizeof(Class_a) 的值是 8Byte。

5) 函数——函数的返回类型所占的空间大小，且函数的返回类型不能是 void。

1.4 指针

1. 指针的概念

为了理解什么是指针，必须先弄清楚数据在内存中是如何存储的，又是如何读取的。如果在程序中定义了一个变量，在编译时就给这个变量分配内存单元。系统根据程序中定义的变量类型，来分配一定长度的空间。例如，C++ 编译系统在 32 位机器上为整型变量分配 4Byte，为单精度浮点型变量分配 4Byte，为字符型变量分配 1Byte。内存区的每一个字节有一个编号，这个编号就是地址，如表 1-1 所示。

表 1-1 用户数据、变量、地址直接的对应关系

地址	用户数据	变量名
...
2000	3	变量 i
2004	6	变量 j
2008	9	变量 k
2012	10	变量 l
...

表 1-1 中展示了用户数据、变量、地址直接的对应关系。假设有变量 i，存的数据是 3，那它在内存中的地址就是 2000。

请务必弄清楚一个内存单元的地址与内存单元的内容这两个概念的区别。其实程序经过编译以后已经将变量名转换为变量的地址，对变量值的存取都是通过地址进行的。这种按变量地址存取变量值的方式称为直接存取方式，或直接访问方式。还可以采用另一种称为间接存取（间接访问）的方式，在程序中定义一种特殊的变量，专门用来存放地址。

由于通过地址能找到所需的变量单元，因此可以说，地址指向该变量单元。因此将地址形象化地称为“指针”，一个变量的地址称为该变量的指针。如果有一个变量是专门用来存放另一变量地址（即指针）的，则它称为指针变量。指针变量的值（即指针变量中存放的值）是地址（即指针）。

指针也是一种变量，普通的变量存放的是实际的数据，而指针变量包含的是内存中的一块地址，这块地址指向某个变量或者函数。指针的内容包括：指针的类型、指针所指向的类型、指针的值以及指针本身所占的内存区。例 1.6 展示了指针的使用。

【例 1.6】 指针使用举例。

```
#include<iostream>
using namespace std;
int main(){
    int p1=1;           // p1 是一个普通的整型变量
    int *p2;            // p2 是一个指针，指向一个整型变量
    p2=&p1;              // 把 p1 的地址赋值给 p2，p2 也就指向了 p1
    cout<<p1<<" "<<*p2<<endl; // *p2 就是取 p2 所指向的地址的内容
```

```

p1=2; // 那么 *p2 的值也是 2
cout<<p1<<" "<<*p2<<endl;
*p2=3; // 那么 p1 的值也是 3
cout<<p1<<" "<<*p2<<endl;
return 0;
}

```

程序的执行结果是：

```

1 1
2 2
3 3

```

例 1.6 中定义了一个整型变量 `p1`，一个指向整型变量的指针 `p2`，并将 `p2` 指向 `p1`。要使用 `p2` 指向的内容，必须在 `p2` 前面加个 `*` 号，也就是 `*p2`。修改了 `p1` 的值，`*p2` 的内容也会跟着改变；同样地，修改 `*p2` 的值，`p1` 的值也会跟着改变。

2. 数组与指针

在 C++ 中，数组名代表数组第一个元素的地址，如下程序定义了两个变量：

```

int *p;
int a[10];

```

若 `p=a` 等价于 `p=&a[0]`，可以通过对 `p`、`a` 的偏移（`int` 类型的指针 `+1` 或 `-1`，是向上或向下偏移 `sizeof(int)` 个 `byte`）来访问数组里的元素，若用 `*(p+i)`、`*(a+i)` 也可以通过传统的数组 `a[i]` 访问各个元素。

(1) 数组指针，也称行指针，具体内容如下所述。

假设有定义 `int (*p)[n]`；且 `()` 优先级高，首先说明 `p` 是一个指针，且指向一个整型的一维数组。这个一维数组的长度是 `n`，也可以说是 `p` 的步长，也就是说执行 `p+1` 时，`p` 要跨过 `n` 个整型数据的长度。如要将二维数组赋给一指针，应这样赋值：

```

int a[3][4];
int (*p)[4]; // 该语句是定义一个数组指针，指向含 4 个元素的一维数组。
p=a; // 将该二维数组的首地址赋给 p，也就是 a[0] 或 &a[0][0]
p++; // 该语句执行过后，也就是 p=p+1；p 跨过行 a[0][ ] 指向了行 a[1][ ]
// 所以数组指针也称指向一维数组的指针，亦称行指针。

```

(2) 指针数组不同于数组指针，具体内容如下所述。

假设有定义 `int *p[n]`；且 `[]` 优先级高，可以理解为先与 `p` 结合成为一个数组，再由 `int*` 说明这是一个整型指针数组，它有 `n` 个指针类型的数组元素。这里若执行 `p+1` 操作则是错误的，`p=a` 这样赋值也是错误的，因为 `p` 是个不可知的表示，只存在 `p[0]`、`p[1]`、`p[2]`...`p[n-1]`，而且它们分别是指针变量，只用来存放变量地址。但可以这样 `*p=a` 赋值这里 `*p` 表示指针数组第一个元素的值，`a` 的首地址的值。

如要将二维数组赋给一指针数组，程序可以是：


```
int *p[3];
int a[3][4];
for(i=0;i<3;i++){
    p[i]=a[i];
}
```

这里 `int *p[3]` 表示一个一维数组内存放着 3 个指针变量，分别是 `p[0]`、`p[1]`、`p[2]`，所以要分别赋值。

这样数组指针和指针数组两者的区别就很明显了：数组指针只是一个指针变量，可以认为是 C 语言里专门用来指向二维数组的，它占用内存中一个指针的存储空间；指针数组是多个指针变量，以数组形式存在内存当中，占用多个指针的存储空间。还需要说明的一点就是，同时用来指向二维数组时，其直接引用和用数组名引用都是一样的。

比如要表示数组中第 *i* 行 *j* 列一个元素，这几种方式都可以：

`*(p[i]+j)`、`*(*(p+i)+j)`、`((p+i))[j]`、`p[i][j]`。

其中，优先级：`()>[]>*`。

3. 字符串与指针

字符串是 C++ 中最经常用到的操作对象之一。用字符数组和字符指针变量都可以实现字符串的存储和运算。例 1.7 展示了字符数组和字符指针是如何使用的。

【例 1.7】 字符数组、字符指针、字符指针数组、字符串变量应用举例。

```
#include<iostream>
#include<string>
using namespace std;
int main(){
    char str[] = "I am a programmer." ; //str 是一个字符数组
    char * str1="abc";                 //str1 是一个字符指针变量，可以指向一个字符串
    char * str2[]={ "hello world", "good bye" };
                                         //str2 是一个字符指针数组，可以存多个字符串
    string str3 = "I am a programmer, too.";
                                         //str3 是一个字符串变量

    cout<<"str: "<<str<<endl;
    cout<<"str1: "<<str1<<endl;
    cout<<"str2[0]: "<<str2[0]<<endl;
    cout<<"str3: "<<str3<<endl;
    return 0;
}
```

程序的执行结果是：

```
str: I am a programmer.
str1: abc
str2[0]: hello world
str3: I am a programmer, too.
```

例 1.7 中，`str` 是一个字符数组，`str1` 是一个字符指针变量，`str2` 是一个字符指针数组，`str3` 是一个字符串变量。

(1) 字符串指针变量本身是一个变量,用于存放字符串的首地址。可以改变 `str1` 使它指向不同的字符串,但不能改变 `str1` 所指的字符串常量。因为定义指针时,编译器并不为指针所指向的对象分配空间,它只是分配指针本身的空间,所以 `abc` 会被当成常量,并且被放到程序的常量区,不能被修改。

(2) 字符串本身是存放在以该首地址为首的一块连续的内存空间中,并以 `'\0'` 作为字符串的结束标志。

(3) 字符数组是由于若干个数组元素组成的,每个元素中存放字符串的一个字符。在定义一个字符数组时,编译后就会分配一个内存单元,每个元素都有确定的地址。

4. 函数与指针

函数指针是指向函数的指针变量。所以,函数指针首先是个指针变量,而且这个变量指向一个函数。C++ 在编译时,每一个函数都有一个入口地址,该入口地址就是函数指针所指向的地址。有了指向函数的指针变量后,就可以用该指针变量调用函数了。

函数指针的声明方法是:

返回值类型 (* 指针变量名) ([形参列表]);

其中,返回值类型说明函数的返回值类型,(* 指针变量名) 这句的括号不能省略。

例如:

```
int func(int a);           // 声明一个函数
int (*f) (int a);         // 声明一个函数指针
f=&func;
```

将 `func` 函数的首地址赋值给函数指针,这里也等价于 `f=&func`; 赋值时函数不带括号,也不带参数,函数名就代表了函数的首地址。例 1.8 展示了函数指针调用函数的方法。

【例 1.8】函数指针使用范例。

```
#include<iostream>
using namespace std;
int Mmin(int x,int y){
    if(x<y)return x;
    return y;
}
int Mmax(int x,int y){
    if(x>y)return x;
    return y;
}
int main(){
    int (*f)(int x,int y);
    int a=10,b=20;
    f=Mmin;           // 把 Mmin 函数的入口地址赋给 f
    cout << (*f)(a,b)<<endl;
    f=Mmax;           // 把 Mmax 函数的入口地址赋给 f
    cout << (*f)(a,b)<<endl;
    return 0;
}
```

程序的执行结果是：

10

20

例 1.8 中定义了一个函数指针 f，两个函数 Mmin 和 Mmax，先后把 f 指向 Mmin 和 Mmax 函数，执行比较两个数，分别得出较小值和较大值。

1.5 引用

1. 引用是什么

对于习惯使用 C 语言进行开发的朋友们，在看到 C++ 中出现的 & 符号后，可能会犯迷糊，虽然在 C 语言中这个符号代表取地址符，但是在 C++ 中它却有着不一样的用途，代表着引用的意思。掌握 C++ 的 & 符号，有利于增强代码质量和提高代码执行效率。

引用是一种变量类型，它用于为一个变量起一个别名。

引用的声明方法是：

类型标识符 & 引用名 = 目标变量名；

假设有一个变量 a，想给它起一个别名 r，可以这样写：

```
int a;
int &r=a;
```

定义引用 r，它是变量 a 的引用，即别名。经过这样的声明后，a 和 r 的作用都一样，都代表着同一变量。a 和 r 占用内存的同一个存储单元，即具有同一地址。在声明一个引用变量时，必须同时使之初始化，即声明它代表哪个变量。函数执行期间，不可以将其再作为其他变量的引用。例 1.9 说明了引用的使用方法。

【例 1.9】引用的使用举例。

```
#include<iostream>
using namespace std;
int main(){
    int a=2;
    int &r=a;
    a=a+4;
    cout<<a<<" "<<r<<endl;    // 因为 a 和 r 的值会同时变化，所以 a 和 r 的值都是 6。
    r=10;
    cout<<a<<" "<<r<<endl;    // r 变了，a 也会变，所以 a 和 r 的值都是 10。
    return 0;
}
```

程序的执行结果是：

6 6

10 10

例 1.9 中展示了引用与变量的关系。r 是 a 的引用，a 变了，r 的值也跟着变；r 变了，a 的值也跟着变。

2. 引用作为参数

引用一个重要的作用就是作为函数的参数。

【例 1.10】 引用作为函数的参数举例。

```
#include<iostream>
using namespace std;
void Mmin1(int a,int b){
    int temp;
    if(a>b){
        temp=a;
        a=b;
        b=temp;
    }
}

void Mmin2(int &a,int &b){           // 引用作为函数的参数
    int temp;
    if(a>b){
        temp=a;
        a=b;
        b=temp;
    }
}

int main(){
    int a=30,b=20;
    Mmin1(a,b);                      //
    cout<<a<<" "<<b<<endl;          // a、b 的值保持不变。
    Mmin2(a,b);
    cout<<a<<" "<<b<<endl;          // a 的值是 20，b 的值是 30。a、b 的值被修改了
    return 0;
}
```

程序的执行结果是：

```
30 20
20 30
```

例 1.10 中定义了两个求最小值的函数，其中 Mmin1 是将一般变量作为函数的参数，Mmin2 则是将引用作为函数的参数。

将一般变量作为函数的参数，传给形参的是变量的值，传递是单向的。如果在执行函数期间形参的值发生变化，并不传回给实参。因为在调用函数时，形参和实参不是同一个存储单元。

使用引用传递函数的参数时，在内存中并没有产生实参的副本，而是对实参直接操作。当使用一般变量传递函数的参数时，当函数发生调用，需要给形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据

较大时，用引用比用一般变量传递参数的效率更高，所占空间更少。

使用指针作为函数的参数虽然也能达到与使用引用同样的效果，但是在被调函数中同样要给形参分配存储单元，且需要重复使用“* 指针变量名”的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参，这些都不太方便。

综上所述，引用是个有效率的选择。

3. 常引用

如果既要提高程序的效率，又要使传递给函数的数据不在函数中被改变，就应该使用常引用。常引用的声明方式是：

`const 类型标识符 &引用名 = 目标变量名；`

用这种方式声明的引用，不能通过引用对目标变量的值进行修改，在程序中使引用的目标成为 `const` 类型，从而保证了引用的安全性，如下所示：

```
int a;
const int &r=a;
r=1;           // 错误
a=1;           // 正确
```

假设有如下函数声明：

```
string func1();
void func2(string &s);
```

那么下面的表达式都是非法的：

```
func2(func1);
func2("hello");
```

原因在于 `func1()` 和 `"hello"` 都将产生一个临时对象，而在 C++ 中，这些临时对象都是 `const` 类型的。因此，上面的表达式就是试图将一个 `const` 类型的对象转化为非 `const` 类型，这是非法的。

引用型参数应该在能被定义成 `const` 的情况下，尽量定义为 `const`。

1.6 结构体、公用体、枚举

1.6.1 结构体、共用体、枚举的概念

1. 结构体的声明方法

结构体的声明方法如下所示：

```
struct 结构名 {
    数据类型 成员名；
```

数据类型 成员名;

};

成员表由若干成员组成,每个成员都是该结构的一个组成部分,对每个成员也必须做类型声明。例 1.11 说明了结构体的使用方法。

【例 1.11】 结构体使用范例。

```
#include<iostream>
#include<string.h>
using namespace std;
struct student{           // 声明一个结构体类型 student
    int num;
    char name[20];
    int age;
};                          // 最后有一个分号
int main(){
    struct student stu1;    // 定义一个 student 类型的变量 stu1
    student stu2;          // 定义时也可以不用 struct
    stu1.num=1;            // 单独对 stu1 的 num 元素赋值
    char temp[20]="Xiao ming";
    strncpy(stu1.name,temp,strlen(temp));
    stu1.age=10;
    cout<<stu1.num<<" "<<stu1.name<<" "<<stu1.age<<endl;
    student *stu3=&stu1;   // stu3 是结构体的指针, 指向 stu1
    (*stu3).num=2;         // stu1 的 num 值被修改成了 2;
    cout<<stu1.num<<" "<<stu1.name<<" "<<stu1.age<<endl;
    return 0;
}
```

程序的执行结果是:

```
1 Xiao ming 10
2 Xiao ming 10
```

例 1.11 中声明了一个结构体类型 student, 定义了两个 student 类型的变量 stu1 和 stu2, 分别用了两种定义方法, 分别是前面有 struct 关键字和没有 struct 关键字。这两种定义方法都是可以的, 显然没 struct 关键字的使用更为方便。

引用结构体变量中成员的一般方式为:

结构体变量名. 成员名

例 1.11 中, stu1.num 表示结构体变量 stu1 中的成员数量的值。

指针也可以应用于结构体, 例 1.11 中 stu3 就是一个结构体变量, 指向 stu1。

2. 共用体

共用体, 用关键字 union 来定义, 它是一种特殊的类。在一个共用体里可以定义多种不同的数据类型, 这些数据共享一段内存, 在不同的时间里保存不同的数据类型和长度的变

量，以达到节省空间的目的。但同一时间只能储存其中一个成员变量的值。

共用体的声明方式为：

```
union 共用体类型名 {  
    数据类型 成员名；  
    数据类型 成员名；  
    ...  
} 变量名；
```

可以使用 union 判断系统是 big endian（大端）还是 little endian（小端）。

【例 1.12】 判断系统是 big endian 还是 little endian。

```
#include<iostream>  
using namespace std;  
union TEST{  
    short a;  
    char b[sizeof(short)];  
};  
int main(){  
    TEST test;  
    test.a=0x0102;// 不能引用共用体变量，只能引用共用体变量中的成员。  
    if(test.b[0]==0x01&&test.b[1]==0x02){  
        cout<<"big endian."<<endl;  
    }  
    else if(test.b[0]==0x02&&test.b[1]==0x01){  
        cout<<"small endian."<<endl;  
    }  
    else{  
        cout<<"unknown"<<endl;  
    }  
    return 0;  
}
```

程序的执行结果是：

small endian.

其中，big endian 是指低地址存放最高有效字节，little endian 则是低地址存放最低有效字节。如果将 0x1234abcd 写入 0x0000 开始的内存中，则结果如表 1-2 所示。

表 1-2 big endian 和 little endian 的内存地址增长方向

地址	big endian	little endian
0x0000	0x12	0xcd
0x0001	0x34	0xab
0x0002	0xab	0x34
0x0003	0xcd	0x12

表 1-2 展示了 0x1234abcd 分别在 big endian 的系统和在 little endian 的系统上的内存地

址是怎么样分布的。例如 0x0000 是 1Byte 的存储单元, 0x12 单元一共占了 8bit, 也就是 1byte, 即占用了 1Byte。

目前, 几乎所有网络协议都是采用 **big endian** 的方式来传输数据的, 当两台采用不同字节序的主机通信时, 在发送数据之前都必须经过字节序的转换成为网络字节序 (**big endian**) 后再进行传播。

3. 枚举

在实际问题中, 有些变量的取值被限定在一个有限的范围内。例如, 一个星期只有 7 天, 一年只有 12 个月, 一个班每周有 6 门课程等。如果把这些量说明为整型, 字符型或其他类型显然是不妥当的。为此, C 语言提供了一种称为“枚举”的类型, 枚举类型在 C++ 中也同样适用。在“枚举”类型的定义中列举出所有可能的取值, 用来说明该“枚举”类型的变量取值不能超过定义的范围。应该说明的是, 枚举类型是一种基本数据类型, 而不是一种构造类型, 因为它不能再分解为任何其他基本类型。

枚举的声明方式为:

```
enum 枚举类型名 { 枚举常量表列 };
```

如同结构和共用体一样, 枚举变量也可用不同的方式说明, 即先定义后说明, 同时定义说明或直接说明。

设有变量 *a, b, c* 是枚举类型 **weekday**, 可采用下述任一种方式:

```
enum weekday{ sun,mou,tue,wed,thu,fri,sat };
enum weekday a,b,c;
```

或者为:

```
enum weekday{ sun,mou,tue,wed,thu,fri,sat }a,b,c;
```

或者为:

```
enum { sun,mou,tue,wed,thu,fri,sat }a,b,c;
```

枚举值是常量, 不是变量。不能在程序中用赋值语句再对它赋值。

例如对枚举 **weekday** 的元素再作以下赋值, 都是错误的:

```
sun=5;
mon=2;
sun=mon;
```

只能把枚举值赋予枚举变量, 不能把元素的数值直接赋予枚举变量, 如下面的语句是正确的:

```
a=sum;
b=mon;
```

而下面的语句则是错误的:


```
a=0;
b=1;
```

如一定要把数值赋予枚举变量，则必须用强制类型转换，如：

```
a=(enum weekday)2;
```

其意义是将顺序号为 2 的枚举元素赋予枚举变量 a，相当于：

```
a=tue;
```

还应该说明的是枚举元素不是字符常量也不是字符串常量，使用时不要加单、双引号。

【例 1.13】enum 使用范例。

```
#include<iostream>
using namespace std;
int main(){
    enum weather{sunny,cloudy,rainy,windy};
    /* 其中 sunny=0,cloudy=1,rainy=2,windy=3,
    默认地，第一个枚举子被赋值为 0 */
    enum fruits{apple=3,orange,banana=7,bear};
    /* 也可以显式地赋值，接下来的枚举子取值是前面一个枚举子的取值 +1，即 orange=4, bear=8 */
    cout<<orange<<endl;
    enum big_cities{guangzhou=1,shenzhen=3, eijing=1,shanghai=2};
    /* 同一枚举中的枚举子的取值不需要是唯一的 */
    return 0;
}
```

程序的执行结果是：

```
4
```

例 1.13 中定义了 3 个枚举类型：weather、fruits 和 big_cities，其中 weather 中的枚举子都被显式地赋值了，fruits 中的枚举子只赋值了其中几个，没显式赋值的则是前面一个枚举子的取值 +1。同一枚举类型中的枚举子的取值不需要是唯一的，可以相同。

1.6.2 结构体、共用体在内存单元占用字节数的计算

一般 64 位机器上各个数据类型所占的存储空间如下所述。

(1) char: 8bit=1byte。

(2) short: 16bit=2byte。

(3) int: 32bit=4byte。

(4) long: 64bit=8byte。

(5) float: 32bit=4byte。

(6) double: 64bit=8byte。

(7) long long: 64bit=8byte。

其中, long 类型在 32 位机器上只占 4Byte, 其他类型在 32 位机器和 64 位机器都是占同样的大小空间。先来看 union 占用内存单元字节数的计算方法。

【例 1.14】 union 的字节数计算。

```
#include<iostream>
using namespace std;
union A{
    int a[5];
    char b;
    double c;
};
int main(){
    cout<<sizeof(A)<<endl;
    return 0;
}
```

程序的执行结果是:

24

union 中变量共用内存, 应以最长的为准, 可是例 1.14 的执行结果却不是预想的 20 (int a[5], $5*4=20\text{Byte}$), 这是因为在共用体内变量的默认内存对齐方式, 必须以最长的 double (8Byte) 对齐, 也就是说应该是 $\text{sizeof}(A)=24$ 。所以将共用体中的 int a[5] 修改成 int a[6] 后, 结果仍然不变; 但如果将 int a[5] 修改成 int a[7], 结果就将变成 32。

再看 struct 的计算方法。

【例 1.15】 struct 的字节数计算。

```
#include<iostream>
using namespace std;
struct B{
    char a;
    double b;
    int c;
}test_struct_b;
int main(){
    cout<<sizeof(test_struct_b)<<endl;
    return 0;
}
```

程序的执行结果是:

24

这是因为 char a 的偏移量为 0, 占用 1Byte; double b 指的是下一个可用的地址的偏移量为 1, 不是 $\text{sizeof}(\text{double})=8$, 需要补足 7Byte 才能使偏移量变为 8; int c 指的是下一个可用的地址的偏移量为 16, 是 $\text{sizeof}(\text{int})=4$ 的倍数, 满足 int 的对齐方式。

故所有成员变量都分配了空间, 空间总的大小为 $1+7+8+4=20$, 不是结构的节边界数

(即结构中占用最大空间的类型所占用的字节数 `sizeof(double)=8`) 的倍数, 所以需要填充 4Byte, 以满足结构的大小为 `sizeof(double)=8` 的倍数, 即 24。

再来看一个混合结构体的大小计算。

【例 1.16】一个混合结构体大小的计算。

```
#include<iostream>
using namespace std;
typedef union{
    long i;
    int k[5];
    char c;
} UDATE;
struct data{
    int cat;
    UDATE cow;
    double dog;
}too;
UDATE temp;
int main(){
    cout<<sizeof(struct data)+sizeof(temp)<<endl;
    return 0;
}
```

假设是测试机器是在 64 位机器上, 那么程序的执行结果应该是多少? UDATE 是一个 union, 作为变量公用空间。里面占用字节数最多的变量是 `int k[5]`, 有 20Byte, 但它要与 long 类型的 8Byte 对齐, 所以占用 24Byte。所以 `sizeof(struct data)` 是 24, temp 是一个 struct, 每个变量分开占用空间, 依次为 `int4+UDATE28+double8=40`, 40 是 4 和 8 的公倍数, 字节已对齐, 故 `sizeof(temp)` 是 40, 所以结果是 `40+24=64`。

1.7 预处理

C++ 提供的预处理功能主要有以下 4 种: 宏定义、文件包含、条件编译和布局控制。文件包含在前面已描述过, 下面重点描述宏定义、条件编译和布局控制, 其中又着重讲述常用宏定义命令、`do...while(0)` 的妙用、条件编译及 `extern"C"` 块的应用知识。

1. 常用宏定义命令

`#define` 命令是一个宏定义命令, 它用来将一个标识符定义为一个字符串, 该标识符被称为宏名, 被定义的字符串称为替换文本。该命令有两种格式: 一种是简单的宏定义, 另一种是带参数的宏定义。

简单的宏定义的声明格式如下所示:

`#define` 宏名 字符串

例: `#define PI 3.1415926`

带参数的宏定义的声明格式如下所示：

#define 宏 (参数表列) 宏

例：#define A(x) x

使用宏定义中，要注意以下问题。

(1) 在简单宏定义的使用中，当替换文本所表示的字符串是一个表达式时，需要加上括号，否则容易引起误解和误用。

【例 1.17】简单宏定义不加括号容易引起误用。

```
#include<iostream>
#define N 2+9
using namespace std;
int main(){
    int a=N*N;
    cout<<a<<endl;
    return 0;
}
```

程序的执行结果是：

29

例 1.17 中就出现了问题：在此程序中存在着宏定义命令，宏 N 代表的字符串是 2+9，在程序中有对宏 N 的使用，一般同学在读该程序时，容易产生的问题是先求解 N 为 2+9=11，然后在程序中计算 a 时使用乘法，即 $N*N=11*11=121$ ，其实该题的结果为 29，为什么结果有这么大的偏差？因为宏展开是在预处理阶段完成的，这个阶段把替换文本只是看作一个字符串，并不会有任何的计算发生，在展开时是在宏 N 出现的地方只是简单地使用串 2+9 来代替 N，并不会增添任何的符号，所以对该程序展开后的结果是 $a=2+9*2+9$ ，计算后结果为 29。要程序如之前想要的结果，只需要写成 #define N (2+9)，即加上括号就行。

(2) 类似地，在带参数的宏定义的使用中，也容易引起误解。例如当需要使用宏替换来求任何数的平方，这时就需要使用参数，以便在程序中用实际参数来替换宏定义中的参数。初学者容易写成如例 1.18 中的形式。

【例 1.18】带参数的宏定义不加括号容易引起误用。

```
#include<iostream>
using namespace std;
#define area(x) x*x
int main()
{
    int y = area(2+2);
    cout<<y<<endl;
    return 0;
}
```

程序的执行结果是：

8

表面上看，给的参数是 $2+2$ ，所得的结果应该为 $4*4=16$ ，但该程序的实际结果为 8。宏定义中要遵循先替换后计算的原则，在上面的程序里， $2+2$ 即为宏 `area` 中的参数，应该由它来替换宏定义中的 `x`，即替换成 $2+2*2+2=8$ 了。那如果遵循 (1) 中的解决办法，把 $2+2$ 括起来，即把宏体中的 `x` 括起来，是否可以解决呢？`#define area(x) (x)*(x)`，对于 `area(2+2)`，替换为 $(2+2)*(2+2)=16$ ，可以解决，但是对于 `area(2+2)/area(2+2)` 又会怎么样呢，有人一看到这道题马上给出结果 1，因为分子分母一样，那么这样就又错了。遵循先替换再计算的规则，这道题替换后会变为 $(2+2)*(2+2)/(2+2)*(2+2)$ 即 $4*4/4*4$ 按照乘除运算规则，结果为 $16/4*4=4*4=16$ 。解决这类问题的方法是在整个宏体上再加一个括号，即 `#define area(x) ((x)*(x))`，不要觉得这没必要，没有它是不行的。

要想能够真正使用好宏定义，在读别人的程序时，一定要记住先将程序中对宏的使用全部替换成它所代表的字符串，不要自作主张地添加任何其他符号，完全展开后再进行相应的计算，就不会求错运行结果。

如果是自己在编程时使用宏替换，则在使用简单宏定义时，当字符串中不只一个符号时，加上括号表现出优先级，如果是带参数的宏定义，则要给宏体中的每个参数加上括号，并在整个宏体上再加一个括号。

2. do...while(0) 的妙用

大家都知道，`do{...}while(condition)` 可以表示循环，但你有没有遇到在一些宏定义中可以不用循环的地方，也用到了 `do{...}while`，比如有这样的宏：

```
#define Foo(x) do{\
    statement one;\
    statement two;\
}while(0)           // 这里没有分号
```

粗看会觉得很奇怪，既然循环里面只执行了一次，那要这个看似多余的 `do...while(0)` 有什么意义呢？再来看这样的宏：

```
#define Foo(x) {\
    statement one;\
    statement two;\
}
```

这两个看似一样的宏，其实是不一样的。前者定义的宏是一个非复合语句，而后者却是一个复合语句。假如有这样的使用场景：

```
if(condition)
    Foo(x);
else
    ...;
```

因为宏在预处理的时候会直接被展开，采用第 2 种写法，会变成：

```

if(condition)
    statement one;
    statement two;
else
    ...///

```

这样会导致 else 语句孤立而出现编译错误。加了 do{...}while(0)，就使得宏展开后，仍然保留初始的语义，从而保证程序的正确性。

3. 条件编译

一般情况下，源程序中所有行的语句都参加编译。但是有时程序员希望其中一部分内容只在满足一定条件时才进行编译，也就是对一部分内容指定编译的条件，这就用到了“条件编译”。

条件编译命令最常见的形式为：

```

#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif

```

它的作用是：当标识符已经被定义过（一般是用 #define 命令定义），则对程序段 1 进行编译，否则编译程序段 2。其中 #else 部分也可以没有，即：

```

#ifdef 标识符
    程序段 1
#endif

```

下面这样的形式则是当指定的表达式值为真（非零）时就编译程序段 1，否则编译程序段 2。可以事先给定一定条件，使程序在不同的条件下执行不同的功能。

```

#if 表达式
    程序段 1
#else
    程序段 2
#endif

```

这里的“程序段”可以是语句组，也可以是命令行。

有时候程序中的某些调试代码，只需要在调试的时候被编译，而不希望在程序的正式发行版中被编译，你可能会看到类似例 1.19 这样的代码段。

【例 1.19】 调试代码巧用条件编译。

```

#include<iostream>
using namespace std;
#define _DEBUG_
int main(){
    int x=10;

```



```

#ifdef _DEBUG_
    cout<<"File:"<< __FILE__<<"Line:"<< __LINE__<<"x:"<<x<<endl;
#else
    printf("x = %d\n", x);
    cout<<x<<endl;
#endif
return 0;
}

```

程序的运行结果是：

```
File:test.cpp,Line:7,x:10
```

当 `_DEBUG_` 没有被定义的时候，仅编译 `#else` 与 `#endif` 之间的代码；当定义了 `_DEBUG_` 符号之后，则会编译 `#ifdef` 与 `#else` 之间的代码。要想定义一个符号很简单，只需要在文件头部加上像这样的一条语句：

```
#define _DEBUG_
```

用 `#define` 命令的目的不在于用 `_DEBUG_` 代表一个字符串，而只是表示已定义过 `_DEBUG_`，因此 `_DEBUG_` 后面写什么字符串都无所谓，甚至可以不写字符串。

4. extern "C" 块的应用

经常能在 C 与 C++ 混编的程序中看到这样的语句：

```

#ifdef __cplusplus
    extern "C" {
#endif
...
#ifdef __cplusplus
}
#endif

```

其中，`__cplusplus` 是 C++ 的预定义宏，表示当前开发环境是 C++。在 C++ 语言中，为了支持重载机制，在编译生成的汇编代码中，会对函数名字进行一些处理（通常称为函数名字改编），如加入函数的参数类型或返回类型等，而在 C 语言中，只是简单的函数名字而已，并不加入其他信息，如下所示：

```

int func(int demo);
int func(double demo);

```

C 语言无法区分上面两个函数的不同，因为 C 编译器产生的函数名都是 `_func`，而 C++ 编译器产生的名字则可能是 `_func_Fi` 和 `_func_Fd`，这样就很好地把函数区别开了。

所以，在 C/C++ 混合编程的环境下，`extern "C"` 块的作用就是告诉 C++ 编译器这段代码要按 C 标准编译，以尽可能地保持 C++ 与 C 的兼容性。例 1.20 说明了 `__cplusplus` 的使用方法。

【例 1.20】 __cplusplus 的使用方法。

```
#include<stdio.h>
int main() {
    #define TO_LITERAL(text) TO_LITERAL_(text)
    #define TO_LITERAL_(text) #text
    #ifndef __cplusplus
        /* this translation unit is being treated as a C one */
        printf("a C program\n");
    #else
        /*this translation unit is being treated as a C++ one*/
        printf("a C++ program\n__cplusplus expands to \"%s\"
            TO_LITERAL(__cplusplus) "\\n");
    #endif
    return 0;
}
```

程序的执行结果是：

```
a C++ program
__cplusplus expands to "1"
```

例 1.20 中程序的意思是：如果没有定义 __cplusplus，那么当前源代码就会被当作 C 源代码处理；如果定义了 __cplusplus，那么当前源代码会被当作 C++ 源代码处理，并且输出 __cplusplus 宏被展开后的字符串。

1.8 本章小结

本章抛砖引玉地讲述了 C++ 中的常用技术，通过简单的例子，读者可轻易学会其使用方法。读者在实际编程时，简单的知识点直接查阅本章即可。

学习 C++，既要会利用 C++ 进行面向过程的结构化程序设计，也要会利用 C++ 进行面向对象的程序设计。接下来的第 2 章，我们将学习面向对象的 C++。



Chapter 2

第 2 章

面向对象的 C++

学习 C++，一定要学会面向对象编程。首先讲下“面向对象”产生的历史原因，主要有以下两点。

(1) 计算机只会按照人所写的代码，一步一步地执行下去，最终得到结果。无论程序多么复杂，计算机总是能轻松应付。结构化编程，就是按照计算机的思维写出的代码，但是人看到这么复杂的逻辑，无法进行维护和扩展。

(2) 结构化设计是以功能为目标来构造应用系统，这种做法导致程序员设计程序时，不得不将客体所构成的现实世界映射到由功能模块组成的解空间中，这种转换过程，背离了人们观察和解决问题的基本思路。

可见，结构化设计在构造系统的时候，无法解决重用、维护、扩展的问题，而且会导致逻辑过于复杂，代码晦涩难懂。于是人们就想，能不能让计算机直接模拟现实的环境，用人类解决问题的思路、习惯、步骤来设计相应的应用程序？这样的程序，人们在读它的时候，会更容易理解，也不需要再把现实世界和程序世界之间来回做转换。于是面向对象的编程思想就产生了。

本章主要从面向对象的封装、继承和多态三大特征来带读者进入面向对象的 C++ 世界。

2.1 类与对象

1. 类与对象的概念

面向对象编程的主要思想是把构成问题的各个事务分解成各个对象，建立对象的目的不

是为了完成一个步骤，而是为了描述一个事物在解决问题中经过的步骤和行为。对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性。类，是创建对象的模板，一个类可以创建多个相同的对象；对象，是类的实例，是按照类的规则创建的。类与对象的关系如图 2-1 所示。

图 2-1 中，人和学生属于类，张三和学生李四都是对象，姓名、身高、地址、年龄、性别、血型都是人这一类的属性，跑步、吃饭，这都是人这个方法。属性是一个变量，用来表示一个对象的特征；方法是一个函数，用来表示对象的操作。对象的属性和方法统称为对象的成员。

每一个实体都是对象，有一些对象是具有相同的结构和特性的。每个对象都属于一个特定的类型，这个特定的类型称为类。正如结构体类型和结构体变量一样，需要先声明一个结构体类型，再用它去定义结构体变量。在 C++ 中也是先声明一个类的类型，然后用它去定义若干个同类型的对象。可以说，对象是类类型的一个变量，类则是对象的模板。类是抽象的，不占用存储空间的；而对象是具体的，占用存储空间。

类类型的声明形式如下：

```
class 类名 {                                // class, 声明一个类必须有的关键字
    private:
        私有的数据和成员函数；
    public:
        公用的数据和成员函数；
};                                           // 类的声明以分号结束
```

其中，private 和 public 称为成员访问限定符。

下面再来看下结构体和类的不同之处。例 2.1 展示了在 C++ 中声明一个结构体类型的方法；例 2.2 则是声明一个类的方法。

【例 2.1】声明一个结构体类型。

```
struct SStudent{
    int num;
    char name[20];
    int age;
};
SStudent st_stu1,st_stu2;                // 定义了两个结构体变量
```

【例 2.2】声明一个类。

```
class CStudent{
```

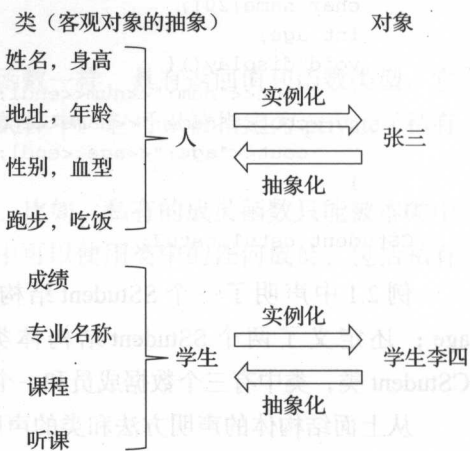


图 2-1 类与对象的关系

```

int num;
char name[20];
int age;           // 这些是数据成员，也称为成员变量
void display(){    // 这是成员函数
    cout<<"num:"<<num<<endl;
    cout<<"name:"<<name<<endl;
    cout<<"age:"<<age<<endl;
}
};
CStudent cstul,cstu2;           // 定义了两个对象

```

例 2.1 中声明了一个 SStudent 结构体，结构体内有三个数据成员：num、name[20] 和 age；还定义了两个 SStudent 结构体类型的变量 st_stu1 和 st_stu1。例 2.2 中声明了一个 CStudent 类，类中有三个数据成员和一个成员函数 display，还定义了两个对象 cstul 和 cstul2。

从上面结构体的声明方法和类的声明方法来看，貌似只有关键字不一样，结构体的关键字是 struct，类的关键字是 class。事实上，声明类的方法是由声明结构体类型的方法发展而来的。但是，struct 中的成员访问权限默认是 public，而 class 中则默认是 private 的。在 C 语言里，struct 中不能定义成员函数，而在 C++ 中，增加了 class 类型后，扩展了 struct 的功能，struct 中也能定义成员函数了。

【例 2.3】 struct 中定义成员函数。

```

struct SStudent{
public:
    void display(){           // 这是成员函数
        cout<<"num:"<<num<<endl;
        cout<<"name:"<<name<<endl;
        cout<<"age:"<<age<<endl;
    }                         // 这里没有分号
private:
    int num;
    char name[20];
    int age;
};

```

例 2.3 中在结构体 SStudent 中定义了一个 display 的 public 的成员函数，3 个 private 的成员变量。请注意，一个成员函数如果在类中定义，在定义结束的 } 之后是不需要加分号的。

在一个类中，关键字 private 和 public 可以分别出现多次，从每个部分的有效范围到出现另一个访问限定符或者类体结束为止。为了使程序清晰，建议大家养成良好的习惯，使每一种成员访问限定符在类体中只出现一次，并且先写 public 部分，把 private 部分放在类体的后部，这样可以使得用户将注意力集中在能被外界调用的成员上，使得阅读者的思路更加清晰。

一个对象的声明方式有以下几种。

(1) class 类名 对象名。

(2) 类名 对象名。

这两种方法是等效的,但显然第二种方法更为简洁与方便。

2. 成员函数

类的成员函数是函数的一种,与第1章介绍过的函数一样,具有返回值和函数类型,它与一般函数的区别在于,它是属于类的成员,出现在类体中。它可以被指定为 `private` (私有的)、`protected` (受保护的) 和 `public` (公用的)。

在使用成员函数时,要注意它的权限以及作用域。比如,私有的成员函数只能被本类中其他成员函数使用,而不能在类外被调用。成员函数中可以使用类中的任何成员,包括私有的和公用的。

成员函数可以在类体中定义,也可以在类外定义。

【例 2.4】 成员函数在类外被定义。

```
class CStudent {
public:
    void display();           // 这里需要分号
private:
    int num;
    char name[20];
    int age;
};

void CStudent::display(){
    cout<<"num:"<<num<<endl;
    cout<<"name:"<<name<<endl;
    cout<<"age:"<<age<<endl;
}
```

例 2.4 中在类 `CStudent` 外定义了 `display` 成员函数。注意,在类外定义成员函数时,必须加上类名,予以限定。“`::`”是作用域限定符或作用域运算符,用它声明函数是属于哪个类的。如果没有写类名或者没有写类名和作用域限定符,则这个函数不属于任何类,而是一个普通函数。成员函数必须先类中声明,然后再在类外定义,即类体的位置应在函数定义之前,否则编译时会出错。

3. 类的封装性

C++ 中通过类实现封装性,把数据和这些数据有关的操作封装在一个类里。但是,人们在使用时,往往不关心类中的具体实现,而只需知道调用哪个函数会得到什么结果,能实现什么功能即可。

为了实现类对象的封装性(数据隐藏和提供访问接口),类类型定义为类成员提供了私有、公有和受保护的 3 种基本访问权限供用户选择,具体内容如下所述。

(1) 私有成员

1) 访问权限:只限于类成员访问。

2) 关键字: `private`。

3) 私有段：从 `private` 关键字开始至其他访问权限声明之间所有成员组成的代码段。

4) 成员种类：数据成员和成员函数。

(2) 公有成员

1) 访问权限：允许类成员和类外的任何访问。

2) 关键字：`public`。

3) 私有段：从 `public` 关键词开始至其他访问权限声明之间所有成员组成的代码段。

4) 成员种类：数据成员和成员函数。

(3) 受保护成员

1) 访问权限：允许类成员和派生类成员访问，不运行类外的任何访问。

2) 关键字：`protect`。

3) 私有段：从 `protect` 关键词开始至其他访问权限声明之间所有成员组成的代码段。

4) 成员种类：数据成员和成员函数。

除了限制访问权限，在写代码时经常要注意“将接口与实现分离”，这也是隐蔽信息的一个重要手段。接口与实现分离，有以下两个好处：①如果想修改或者扩充类的功能，只需要修改类中的实现，类外部分可以不用修改；②如果发现类中数据成员数据有错，则只需要在类内检查访问这些数据成员的成员函数。

一般是将类的声明放在指定的头文件中，用户如果想使用这个类，直接包含这个头文件即可。因为在头文件中有类的声明，所以可以直接在程序中用这个类来定义对象。为了实现信息隐蔽，会把类成员函数的定义放在另一个文件中，而不放在头文件中。

例如，可以将类 `Student` 的声明放在 `student.h` 中。

【例 2.5】类的定义与使用。

`student.h` 中的代码为：

```
class CStudent{
public:
    void display();
private:
    int num;
    char name[20];
    int age;
};
```

`student.cpp` 中的代码为：

```
#include<iostream>
#include "student.h" //这里需要 include 这个头文件，否则无法找到 Student 类
using namespace std;
void CStudent::display(){ //这里要注明是 Student 类的
    cout<<"num:"<<num<<endl;
    cout<<"name:"<<name<<endl;
    cout<<"age:"<<age<<endl;
}
```

main.cpp 中的代码为:

```
#include<iostream>
#include "student.h"           // 注意这里是双引号
int main(){
    CStudent stu1;             // 定义 stu1 对象
    stu1.display();            // 指向 stu1 对象的成员函数
    return 0;
}
```

执行以下这 3 行命令即可编译成功:

```
g++ -c student.cpp
g++ -c main.cpp
g++ -o main main.o student.o
```

编译后, 会生成 main 文件, 执行 ./main 命令, 获得程序的执行结果为:

```
num:0
name:
age:0
```

这种结果是由于还没有对数据成员进行初始化导致的, 下面的一节会介绍如何初始化一个对象。

例 2.5 中定义了一个类 CStudent, 所有的数据成员和成员函数都声明在头文件 student.h 中, 而成员函数的定义则是在 .cpp 文件 student.cpp 中。main.cpp 中要使用 CStudent 类时需要把头文件 student.h 包含进来。编译时, 编译器 g++ 会把 main.cpp 和 student.cpp 分别编译成 main.o 和 student.o, 再把 main.o 和 student.o 链接成可执行文件 main, 图 2-2 展示了这一过程。

编译与链接的相关知识会在第 4 章详细展开, 这里只作简单介绍。

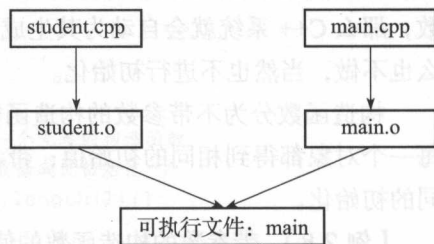


图 2-2 编译与链接

4. 构造函数

数据成员是不能在类中初始化的, 而构造函数, 正是为此而生, 主要用来处理数据成员的初始化。它不需要用户调用, 而是在建立对象时自动执行的。

构造函数的名字必须与类名相同, 而不能由用户任意命名, 以便编译系统能识别它并把它作为构造函数处理。它是一个没有返回值的函数, 构造函数在类中定义如例 2.6 所示。

【例 2.6】构造函数的定义。

```
class Time{
public:
    Time(){                // 这就是构造函数
        hour=0;
        minute=0;
```

```

        second=0;
    }
    set_time();
    get_time();
private:
    int hour,minute,second;
};

```

构造函数也可以在类外定义，如例 2.7 所示。

【例 2.7】 在类外定义构造函数。

```

class Time{
public:
    Time(); // 对构造函数进行声明
    set_time();
    get_time();
private:
    int hour,minute,second;
};

Time::Time(){ // 定义构造函数，需要加上类名和域限定符 "::"
    hour=0;
    minute=0;
    second=0;
}

```

在构造函数的函数体中，不仅可以对数据成员赋值，也可以包含其他语句。不过不提倡在构造函数中加入与初始化无关的内容，以保证程序清晰。如果用户自己没有定义构造函数，那么 C++ 系统就会自动为其生成一个构造函数，只是这个构造函数的函数体是空的，什么也不做，当然也不进行初始化。

构造函数分为不带参数的构造函数与带参数的构造函数。不带参数的构造函数使该类的每一个对象都得到相同的初始值；带参数的构造函数则可以方便地实现对不同的对象进行不同的初始化。

【例 2.8】 带参数的构造函数的使用举例。

```

#include<iostream>
using namespace std;
#define pi 3.1415
class Circle{
public:
    Circle(int r); // 形参列表
    double Area();
private:
    int radius; // 数据成员
};

Circle::Circle(int r){
    radius=r;
}

double Circle::Area(){

```

```

    return pi*radius*radius;
}
int main(){
    Circle cir1(10);
    cout<<"cir1's area: "<<cir1.Area()<<endl;
    Circle cir2(1);
    cout<<"cir2's area: "<<cir2.Area()<<endl;
    return 0;
}

```

程序的执行结果是：

```

cir1's area: 314.15
cir2's area: 3.1415

```

例 2.8 中的构造函数带了一个整型的参数，并在构造函数中将 *r* 参数赋值给了成员变量 *radius*。定义对象时可利用构造函数直接对成员变量赋值。

另外，C++ 还提供另一种初始化数据成员的方法：参数初始化表。这种方法不在函数体内对数据成员初始化，而是在函数首部实现。例 2.8 中的构造函数可以改写为下列形式：

```

Circle::Circle(int r):radius(r){} // 后面没有分号

```

在 C++ 中，一个类可以同时定义多个构造函数，以提供不同的初始化方法。这些构造函数的参数个数不同或参数的类型不同，这就是构造函数的重载。

【例 2.9】 构造函数重载的使用。

```

#include <iostream>
using namespace std;
class Box{
public:
    Box(); // 声明一个无参的构造函数
    /* 声明一个有参的构造函数，并用参数的初始化列表对数据成员初始化 */
    Box(int h,int w,int l):height(h),width(w),length(l){}
    int volume();
private:
    int height,width,length;
};
Box::Box(){ // 定义无参的构造函数
    height=1;
    width=2;
    length=3;
}
int Box::volume(){
    return height*width*length;
}
int main(){
    Box box1; // 不指定实参
    cout<<"box1's volume: "<<box1.volume()<<endl;
    Box box2(2,5,10); // 指定实参
}

```

```

        cout<<"box2's volume: "<<box2.volume()<<endl;
        return 0;
    }

```

程序的执行结果是：

```

box1's Volume: 6
box2's Volume: 100

```

例 2.9 中定义了两个构造函数，一个是无参的，一个是有参的，两个函数的函数名一样，但参数格式不一样，所以是函数重载。若定义对象时不指定实参，则调用无参的构造函数。

调用构造函数时不必给出实参的构造函数，称为默认构造函数。无参的构造函数属于默认构造函数。一个类只能有一个默认构造函数。即使一个类中有多个构造函数，但建立对象时，都只执行其中一个构造函数。

构造函数中的参数的值，可以通过实参传递，也可以指定为某些默认值。

【例 2.10】构造函数默认参数的使用。

```

#include <iostream>
using namespace std;
class Box{
public:
    Box(int h=2,int w=2,int l=2);    // 在声明构造函数时指定默认参数
    int volume();
private:
    int height,width,length;
};
Box::Box(int h,int w,int len){    // 在定义函数时可以不指定默认参数
    height=h;
    width=w;
    length=len;
}
int Box::volume(){
    return height*width*length;
}
int main(){
    Box box1(1);    // 不指定第 2、3 个实参
    cout<<"box1's volume: "<<box1.volume()<<endl;
    Box box2(1,3);    // 不指定第 3 个实参
    cout<<"box2's volume: "<<box2.volume()<<endl;
    return 0;
}

```

程序的执行结果是：

```

box1's volume:4
box2's volume:6

```

例 2.10 中定义了一个带有默认参数的构造函数，是在声明时指定默认参数，而定义时则可以不用指定默认参数。定义对象时，可以传 0 ~ 3 个参数，传了几个参数，就替换前面的几

个参数，其余都还是使用默认参数。

使用默认参数的好处在于：调用构造函数时就算没有提供参数也不会出错，且对每一个对象能有相同的初始化状况。

不过，应该在声明构造函数默认值时指定默认参数值，而不能只在定义构造函数时指定默认参数值。如果构造函数中的参数全指定了默认值，则在定义对象时，可给一个实参或多个实参，也可以不给实参。

一个类中如果定义了全是默认参数的构造函数后，就不能再定义重载构造函数了。假设 Box 类定义了以下 3 个构造函数：

```
Box(int =10,int =10,int =10);
Box();
Box(int,int);
```

若有以下定义语句，思考注释中的问题：

```
Box box1;           // 是调用上面的第一个默认参数的构造函数，还是第二个默认构造函数？
Box box2(15,30);    // 是调用上面的第一个默认参数的构造函数，还是第三个构造函数？
```

5. 析构函数

析构函数的名字是类名的前面加一个“~”符号。在 C++ 中，“~”符号是位取反运算符，类似地，析构函数的作用与构造函数相反。它也不需要用户来调用它，不过，它是在对象声明周期结束时自动执行的。

程序执行析构函数的时机有以下 4 种。

(1) 如果在函数中定义了一个对象，当这个函数调用结束时，对象会被释放，且在对象释放前会自动执行析构函数。

(2) static 局部对象在函数调用结束时对象不释放，所以也不执行析构函数，只有在 main 函数结束或调用 exit 函数结束程序时，才调用 static 局部对象的析构函数。

(3) 全局对象则是在程序流程离开其作用域（如 main 函数结束或调用 exit 函数）时，才会执行该全局对象的析构函数。

(4) 用 new 建立的对象，用 delete 释放该对象时，会调用该对象的析构函数。

析构函数的作用不是删除对象，而是在撤销对象占用的内存前完成一些清理工作，使得这些内存可以供新对象使用。析构函数的作用也不限于释放资源方面，它还可以被用来执行用户希望在最后一次使用对象之后所执行的任何操作。

【例 2.11】 析构函数的使用方法举例。

```
#include<iostream>
using namespace std;
class Box{
public:
    Box(int h=2,int w=2,int l=2);
    ~Box() {
```

// 析构函数


```

        cout<<"Destructor called."<<endl;           // 析构函数里的内容
    }
    int volume(){
        return height*width*length;
    }
private:
    int height,width,length;
};
Box::Box(int h,int w,int len){
    height=h;
    width=w;
    length=len;
}
int main(){
    Box box1;
    cout<<"The volume of box1 is "<<box1.volume()<<endl;
    cout<<"hello."<<endl;
    return 0;
}

```

程序的运行结果：

```

The volume of box1 is 8
hello.
Destructor called.

```

例 2.11 中定义了类 Box 的析构函数，析构函数中输出 Destructur called., 也就是析构函数被执行时就会输出这条消息。由程序的执行结果可以看出，析构函数是对象释放内存前执行的。

如果用户没有编写析构函数，编译系统会自动生成一个默认的析构函数，但不进行任何操作，所以许多简单的类中没有用显式的析构函数。

6. 静态数据成员

有时需要为某个类的所有对象分配一个单一的存储空间。在 C 语言中，可以使用全局变量，但这样很不安全。全局数据可以被任何人修改，而且在一个项目中，它很容易和其他名字冲突。如果可以把数据当成全局变量那样去存储，但又被隐藏在类的内部，而且清楚地与这个类相联系，这种处理方法就是最理想的。这个可以用类的静态数据成员来实现。类的静态成员拥有一块单独的存储区，而不管创建了多少个该类的对象。所有这些对象的静态数据成员都共享这一块静态存储空间，这就为这些对象提供了一种互相通信的方法。静态数据成员是属于类的，它只在类的范围内有效，可以是 public、private 或 protected 的范围。

因为不管产生了多少对象，类的静态数据成员都有着单一的存储空间，所以存储空间必须定义在一个单一的地方。如果一个静态数据成员被声明而没有被定义，链接器会报告一个错误：“定义必须出现在类的外部而且只能定义一次”。因此静态数据成员的声明通常会放在一个类的实现文件中。举例如下所示。

xxx.h 类型文件中:

```
class base{
public:
    static int var;           // 声明静态数据成员
};
```

xxx.cpp 类型文件中:

```
int base::var=10;           // 定义静态数据成员, 不必在初始化语句里加上 static
```

在头文件中定义(初始化)静态成员容易引起重复定义的错误, 比如这个头文件同时被多个 .cpp 文件所包含的时候。即使加上 `#ifndef #define #endif` 或者 `#pragma once` 也不行。

C++ 静态数据成员被类的所有对象所共享, 包括该类的派生类的对象。派生类对象与基类对象共享基类的静态数据成员。静态的数据成员在内存中只占一份空间。如果改变它的值, 则在各个对象中这个数据成员的值同时都改变了, 这样可以节约空间, 提高效率。下面程序展示了修改基类的静态数据成员, 同时影响派生类对象和基类对象的情况。

【例 2.12】 静态的数据成员基类和派生类对象共享。

```
#include<iostream>
using namespace std;
class Base{
public:
    static int var;
};
int Base::var=10;
class Derived:public Base{
};
int main(){
    Base base1;
    base1.var++;           // 通过对象名引用
    cout<<base1.var<<endl; // 输出 11
    Base base2;
    base2.var++;
    cout<<base2.var<<endl; // 输出 12
    Derived derived1;
    derived1.var++;
    cout<<derived1.var<<endl; // 输出 13
    Base::var++;           // 通过类名引用
    cout<<derived1.var<<endl; // 输出 14
    return 0;
}
```

程序的执行结果是:

```
11
12
13
14
```

例 2.12 中在基类 Base 中定义了一个静态数据成员 var，类 Derived 继承了类 Base。无论是通过基类对象，或者是派生类对象，都可以改变静态数据成员 var 的值。

如果只声明了类而未定义对象，类的一般数据成员是不占内存空间的，只有在定义对象时才会为对象的数据成员分配空间。但是静态数据成员不属于某一个对象，所以在为对象所分配的空间中不包括静态数据成员所占的空间，静态数据成员是在所有对象之外单独开辟一段空间来存放。只要在类中定义了静态数据成员，即使不定义对象，也为静态数据成员分配了空间，它可以被引用。

在一个类中可以有一个或多个静态数据成员，所有对象都共享这些静态数据成员，都可以引用它。

如果在一个函数中定义了静态变量，在函数结束时该静态变量并不被释放，仍然存在并保留其值。静态数据成员也类似，它不随对象的建立而分配空间，也不随对象的撤销而释放。静态数据成员是程序在编译时被分配空间，到程序结束时释放空间。

静态数据成员可以通过对象名引用，也可以通过类名来引用。

7. 静态成员函数

与数据成员类似，成员函数也可以定义为静态的，在类中声明函数的前面加 static 关键字就成了静态成员函数，如：

```
static int volume();
```

和静态数据成员一样，静态成员函数也是类的一部分，而不是对象的一部分。如果要在类外调用公用的静态成员函数，要用类名和域运算符“::”，如：

```
Box::volume();
```

实际上也允许通过对象名调用静态成员函数，如：

```
a.volume();
```

但这并不意味着此函数是属于对象 a 的，而只是用 a 的类型而已。

与静态数据成员不同，静态成员函数的作用不是为了对象之间的沟通，而是为了能处理静态数据成员。

当调用一个对象的成员函数（非静态成员函数）时，系统会把该对象的起始地址赋给成员函数的 this 指针。而静态成员函数并不属于某一对象，它与任何对象都无关，因此静态成员函数没有 this 指针。既然它没有指向某一对象，也就无法对一个对象中的非静态成员进行默认访问（即在引用数据成员时不指定对象名）。

可以说，静态成员函数与非静态成员函数的根本区别是：非静态成员函数有 this 指针，而静态成员函数没有 this 指针。由此决定了静态成员函数不能访问本类中的非静态成员。

静态成员函数可以直接引用本类中的静态数据成员，因为静态成员同样是属于类的，可以直接引用。在 C++ 程序中，静态成员函数主要用来访问静态数据成员，而不访问非静态成员。

假如在一个静态成员函数中有以下语句：

```
cout<<height<<endl;    // 若 height 已声明为 static, 则引用本类中的静态成员, 合法
cout<<width<<endl;     // 若 width 是非静态数据成员, 不合法
```

但是, 并不是绝对不能引用本类中的非静态成员, 只是不能进行默认访问, 因为无法知道应该去找哪个对象。如果一定要引用本类的非静态成员, 应该加对象名和成员运算符“.”, 如:

```
cout<<a.width<<endl;    // 引用本类对象 a 中的非静态成员
```

假设 a 已定义为 Box 类对象, 且在当前作用域内有效, 则此语句合法。不过, 最好养成这样的习惯: 只用静态成员函数引用静态数据成员, 而不引用非静态数据成员。这样思路更清晰、逻辑更清楚, 不易出错。

【例 2.13】静态成员函数的使用方法举例。

```
#include<iostream>
using namespace std;
class CStudent{
public:
    CStudent (int n,int s):num(n),score(s){}    // 定义构造函数
    void total();
    static double average();
private:
    int num;
    int score;
    static int count;
    static int sum;    // 这两个数据成员是所有对象共享的
};
int CStudent::count=0;    // 定义静态数据成员
int CStudent::sum=0;
void CStudent::total(){    // 定义非静态成员函数
    sum+=score;    // 非静态数据成员函数中可使用静态数据成
    // 员、非静态数据成员
    count++;
}
double CStudent::average(){    // 定义静态成员函数
    return sum*1.0/count;    // 可以直接引用静态数据成员, 不用加类名
}
int main(){
    CStudent stu1(1,100);
    stu1.total();    // 调用对象的非静态成员函数
    CStudent stu2(2,98);
    stu2.total();
    CStudent stu3(3,99);
    stu3.total();
    cout<< CStudent::average()<<endl;    // 调用类的静态成员函数, 输出 99
}
```

程序的执行结果是：

99

例 2.13 中声明了一个 CStudent 类，类中有静态成员函数 average、静态数据成员 count 和 sum。静态数据成员 count 和 sum 必须在类的外部定义，在非静态成员函数 total 中可以使用静态数据成员、非静态数据成员，而在静态成员函数中则可以不加类名直接引用静态数据成员。

8. 对象的存储空间

很多 C++ 书籍中都介绍过一个对象需要占用多大的内存空间，最权威的结论是：非静态成员变量总和加上编译器为了 CPU 计算做出的数据对齐处理和支持虚函数所产生的负担的总和。下面分别看看数据成员、成员函数、构造函数、析构函数、虚函数的空间占用情况。

先来看看一个空类的存储空间是多少个 Byte 呢？可以看例 2.14 的程序。

【例 2.14】空类存储空间的计算。

```
#include<iostream>
using namespace std;
class CBox{
};
int main(){
    CBox boxobj;
    cout<<sizeof(boxobj)<<endl;// 输出 1
    return 0;
}
```

程序的执行结果是：

1

例 2.14 中定义了一个空类 CBox，里面既没有数据成员，也没有成员函数。程序执行结果显示它的大小为 1。

空类型对象中不包含任何信息，应该大小为 0。但是当声明该类型的对象的时候，它必须在内存中占有一定的空间，否则无法使用这些对象。至于占用多少内存，由编译器决定。C++ 中每个空类型的实例占 1Byte 空间。

【例 2.15】只有成员变量的类的存储空间计算。

```
#include<iostream>
using namespace std;
class CBox{
    int length,width,height;
};
int main(){
    CBox boxobj;
    cout<<sizeof(boxobj)<<endl;
```



```

    return 0;
}

```

程序的执行结果是：

```
12
```

例 2.15 中，类 CBox 中只有 3 个成员变量，由于整型变量占 4Byte，所以对象所占的空间就是 12Byte。那静态成员变量是否也占存储空间呢？

【例 2.16】 有成员变量和静态成员变量的类的存储空间计算。

```

#include<iostream>
using namespace std;
class CBox{
    int length,width,height;
    static int count;
};
int main(){
    CBox boxobj;
    cout<<sizeof(boxobj)<<endl;
    return 0;
}

```

程序的执行结果是：

```
12
```

例 2.16 中，类 CBox 中有 3 个普通数据成员和 1 个静态数据成员，比例 2.14 中多了一个静态数据成员，但是程序的执行结果还是 12，也就证明了静态数据成员是不占对象的内存空间的。

【例 2.17】 类中只有 1 个成员函数的存储空间计算。

```

#include<iostream>
using namespace std;
class CBox{
    int foo();
};
int main(){
    CBox boxobj;
    cout<<sizeof(boxobj)<<endl;
    return 0;
}

```

程序的执行结果是：

```
1
```

例 2.17 中类 CBox 中只有一个成员函数，类 CBox 的对象 boxobj 的大小却只有 1Byte，和空类对象是一样的，所以可以得出，成员函数是不占空间的。

【例 2.18】 类中构造函数、析构函数的空间占用情况。

```
#include<iostream>
using namespace std;
class CBox{
public:
    CBox(){};
    ~CBox(){};
};
int main(){
    CBox boxobj;
    cout<<sizeof(boxobj)<<endl;
    return 0;
}
```

程序的执行结果是：

1

例 2.18 中类 CBox 中只有构造函数和析构函数，类 CBox 的对象 boxobj 的大小也只有 1Byte，和空类对象是一样的，所以可以得出，构造函数和析构函数也是不占空间的。

【例 2.19】 类中有虚的析构函数的空间计算。

```
#include<iostream>
using namespace std;
class CBox{
public:
    CBox(){};
    virtual ~CBox(){};
};
int main(){
    CBox boxobj;
    cout<<sizeof(boxobj)<<endl;    // 输出 4
    return 0;
}
```

程序的执行结果是：

8

例 2.19 中，类 CBox 中有 1 个构造函数和 1 个虚的析构函数，程序的执行结果是 8。事实上，编译器为了支持虚函数，会产生额外的负担，这正是指向虚函数表的指针的大小。（指针变量在 64 位的机器上占 8Byte。）如果一个类中有一个或者多个虚函数，没有成员变量，那么类相当于含有一个指向虚函数表的指针，占 8Byte。

【例 2.20】 继承空类和多重继承空类存储空间计算。

```
#include<iostream>
using namespace std;
class A{
```

```

};
class B{
};
class C:public A{
};
class D:public virtual B{
};
class E:public A,public B{
};
int main(){
    A a;
    B b;
    C c;
    D d;
    E e;
    cout<<"sizeof(a):"<<sizeof(a)<<endl;
    cout<<"sizeof(b):"<<sizeof(b)<<endl;
    cout<<"sizeof(c):"<<sizeof(c)<<endl;
    cout<<"sizeof(d):"<<sizeof(d)<<endl;
    cout<<"sizeof(e):"<<sizeof(e)<<endl;
    return 0;
}

```

程序的执行结果是:

```

sizeof(a):1
sizeof(b):1
sizeof(c):1
sizeof(d):8
sizeof(e):1

```

例 2.20 中定义了一个空类 A 和 B, 类 C 继承了类 A, 类 D 继承了虚基类 B, 类 E 继承了类 A 和类 B。这些类的对象所占的空间都是 1Byte。由此可见, 单一继承的空类空间也是 1, 多重继承的空类空间还是 1, 但是虚继承涉及虚表 (虚指针), 所以 `sizeof(d)=8`。

综上所述, 每个对象所占用的存储空间只是该对象的非静态数据成员的总和, 其他都不占用存储空间, 包括成员函数和静态数据成员。函数代码是存储在对象空间之外的, 而且, 函数代码段是公用的, 即如果对同一个类定义了 10 个对象, 这些对象的成员函数对应的是同一个函数代码段, 而不是 10 个不同的函数代码段。

9. this 指针

每个对象中的数据成员都分别占有存储空间, 如果对同一个类定义了 n 个对象, 则有 n 组同样大小的空间以存放 n 个对象中的数据成员。不同对象都调用同一个函数代码段。那么, 当不同对象的成员函数引用数据成员时, 怎么能保证所引用的是所指定的对象的数据成员呢?

假设, 对于上述例子中定义的 Box 类, 定义了 3 个同类对象 a、b、c。如果有 `a.volume()`, 应该是引用对象 a 中的 `height`、`width` 和 `length`, 以计算出箱子 a 的体积; 如果有 `b.volume()`,

应该是引用对象 **b** 中的 **height**、**width** 和 **length**，计算出箱子 **b** 的体积。而现在都用同一个函数段，系统怎样使它分别引用 **a** 或 **b** 中的数据成员呢？

在每一个成员函数中都包含一个特殊的指针，这个指针的名字是固定的，称为 **this** 指针。它是指向本类对象的指针，它的值是当前被调用的成员函数所在的对象的起始地址。例如，当调用成员函数 **a.volume** 时，编译系统就把对象 **a** 的起始地址赋给 **this** 指针，在成员函数引用数据成员时，就按照 **this** 的指向找到对象 **a** 的数据成员。例如 **volume** 函数要计算 **height*width*length** 的值，实际上是执行：

```
(this->height)*(this->width)*(this->length)
```

由于当前 **this** 指向 **a**，因此相当于执行：

```
(a.height)*(a.width)*( a.length)
```

这就计算出箱子 **a** 的体积。同样如果有 **b.volume()**，编译系统就把对象 **b** 的起始地址赋给成员函数 **volume** 的 **this** 指针，显然计算出来的是箱子 **b** 的体积。

this 指针是隐式使用的，它是作为参数被传递给成员函数。本来，成员函数 **volume** 的定义如下：

```
int Box::volume(){
    return (height*width*length);
}
```

C++ 把它处理为：

```
int Box::volume(Box *this){
    return (this->height * this->width * this->length);
}
```

即在成员函数的形参表列中增加一个 **this** 指针。在调用该成员函数时，实际上是用以下方式调用的：

```
a.volume(&a);
```

将对象 **a** 的地址传给形参 **this** 指针，然后按 **this** 的指向去引用其他成员。


需要说明的是，这些都是编译系统自动实现的，不必人为地在形参中增加 **this** 指针，也不必将对象 **a** 的地址传给 **this** 指针，但在需要时也可以显式地使用 **this** 指针。

例如在 **Box** 类的 **volume** 函数中，下面两种表示方法都是合法的、相互等价的：

```
return (height * width * length);           // 隐含使用 this 指针
return (this->height * this->width * this->length); // 显式使用 this 指针
```

可以用 ***this** 表示被调用的成员函数所在的对象，***this** 就是 **this** 所指向的对象，即当前的对象。例如在成员函数 **a.volume()** 的函数体中，如果出现 ***this**，它就是对象 **a**。上面的 **return** 语句也可写成：

```
return((*this).height * (*this).width * (*this).length);
```

 **注意** *this 两侧的括号不能省略，不能写成 *this.height。因为成员运算符“.”的优先级高于指针运算符“*”，因此，* this.height 就相当于 * (this.height)，而 this.height 是不合法的，编译会出错。

所谓“调用对象 a 的成员函数 f”，实际上是在调用成员函数 f 时使 this 指针指向对象 a，从而访问对象 a 的成员。在使用“调用对象 a 的成员函数 f”时，应当对它的含义有正确的理解。

this 指针有以下特点。

- (1) 只能在成员函数中使用，在全局函数、静态成员函数中都不能使用 this。
- (2) this 指针是在成员函数的开始前构造，并在成员函数的结束后清除。
- (3) this 指针会因编译器不同而有不同的存储位置，可能是栈、寄存器或全局变量。
- (4) this 是类的指针。
- (5) 因为 this 指针只有在成员函数中才有定义，所以获得一个对象后，不能通过对象使用 this 指针，所以也就无法知道一个对象的 this 指针的位置。不过，可以在成员函数中指定 this 指针的位置。
- (6) 普通的类函数（不论是非静态成员函数，还是静态成员函数）都不会创建一个函数表来保存函数指针，只有虚函数才会被放到函数表中。

10. 类模板

有时，两个或多个类的功能是相同的，但仅仅因为数据类型不同，就要分别定义两个类，如下面的例 2.21 声明了一个类。

【例 2.21】 操作整数的类。

```
class Operation_int{
public:
    Operation_int(int a,int b):x(a),y(b){}
    int add(){
        return x+y;
    }
    int subtract(){
        return x-y;
    }
private:
    int x,y;
};
```

这个类的作用是两个整数的加减，如果要对两个浮点数做加减，就只得新定义一个类，比如例 2.22 所示。

【例 2.22】 操作浮点数的类。

```
class Operation_double{
```

```

public:
    Operation_double(double a, double b):x(a),y(b){}
    double add(){
        return x+y;
    }
    double subtract(){
        return x-y;
    }
private:
    double x,y;
};

```

因为参数的类型不同，所以不能复用，这也使得代码量剧增。为了解决这类问题，C++中提供了类模板的功能。可以先声明一个通用的类模板，这个类模板可以有一个或多个虚拟的类型参数，对以上例子中的两个类，可以定义如例 2.23 这样的类模板。

【例 2.23】操作两个数的类模板。

```

template<class T>           // 声明一个模板，虚拟类型名为 T
class Operation {
public:
    Operation (T a, T b):x(a),y(b){}
    T add(){
        return x+y;
    }
    T subtract(){
        return x-y;
    }
private:
    T x,y;
};

```

例 2.23 这个类模板与上面的类相比，有以上两个不同点。

(1) 声明类模板时增加了下面这一行代码：

```
template <class 类型参数名 >
```

其中，**template** 是声明类模板时必须写的关键字，意思是“模板”。关键字 **class** 后的类型参数名可以是任意的合法标识符，本例中 **T** 就是一个类型参数名。

(2) 原有的类型名 **int** 换成虚拟类型参数名 **T**。

在建立类对象时，如果将实际类型指定为 **int** 型，编译系统就会用 **int** 取代 **T**；如果指定为 **double**，编译系统就会用 **double** 取代 **T**。

声明一个类模板的对象时，要用实际类型名去取代虚拟的类型，这样才能使它变成一个实际的对象，如：

```
Operation <int> opobj(1,2);
```

在类模板名之后的尖括号里指定实际的类型名，这样在编译时，编译系统就用 **int** 取代类模板中的类型参数 **T**，这样就把类模板具体化了，或者说实际化了。例 2.24 中实现了一个

类模板，利用它可以分别对两个整数和两个浮点数进行加、减操作。

【例 2.24】 用类模板实现对两个数的加、减操作。

```
#include <iostream>
using namespace std;
template<class T>
class Operation {
public:
    Operation (T a, T b):x(a),y(b){}
    T add(){
        return x+y;
    }
    T subtract(){
        return x-y;
    }
private:
    T x,y;
};

int main(){
    Operation <int> op_int(1,2);
    cout<<op_int.add()<<" "<<op_int.subtract()<<endl; // 输出 3、-1
    Operation <double> op_double(1.2,2.3);
    cout<<op_double.add()<<" "<<op_double.subtract()<<endl; // 输出 3.5、-1.1
    return 0;
}
```

程序的执行结果是：

```
3 -1
3.5 -1.1
```

例 2.24 中声明了一个类模板，可以对两个相同类型的数进行加、减操作：如果是传入两个整数，则对两个整数进行加减；如果传入两个浮点数，则对两个浮点数进行加减。

如果类模板的成员函数是在类外定义的，则需要这么写：

```
template<class T>
T Operation <T> :: add(){
    return x+y;
}
```

综上所述，可以这样声明和使用类模板。

- (1) 先写一个实际的类。
- (2) 将此类中准备改变的类型名改用一个自己指定的虚拟类型名。
- (3) 在类声明前面加入一行，格式为：template<class 虚拟类型参数>。
- (4) 用类模板定义对象时用以下形式：

类模板名 < 实际类型名 > 对象名；
类模板名 < 实际类型名 > 对象名 (实参表列)；

(5) 如果在类模板外定义成员函数，应写成类模板形式：

```
template <class 虚拟类型参数>
```

```
函数类型 类模板名 <虚拟类型参数>::成员函数名(函数形参表列) {…}
```

类模板是对一批仅数据成员类型不同的类的抽象，只要为这一批类所组成的整个类家族创建一个类模板，即给出一套程序代码，就可以用来生成多种具体的类，从而大大提高编程的效率。

11. 析构函数与构造函数的执行顺序

前面讲了析构函数执行的时机，下面再来对比下构造函数和析构函数的调用时间和调用顺序。首先看下面包含构造函数和析构函数的 C++ 程序的执行结果，以此来判断二者执行的顺序。

【例 2.25】构造函数和析构函数的执行顺序实例。

```
#include<iostream>
using namespace std;
class CBox{
public:
    CBox (int h,int w,int l){
        height=h;
        width=w;
        length=l;
        cout<<"Constructor called."<<endl;           // 构造函数被执行时输出信息
    }
    ~CBox (){
        cout<<"Destructor called."<<length<<endl;    // 析构函数
        // 析构函数被执行时输出
    }
    int volume(){
        return height*width*length;
    }
private:
    int height,width,length;
};
int main(){
    CBox box1(1,2,3);
    cout<<box1.volume()<<endl;
    CBox box2(2,3,4);
    cout<<box2.volume()<<endl;
    return 0;
}
```

程序的执行结果为：

```
Constructor called.
```

```
6
```

```
Constructor called.
```

```
24
```

```
Destructor called.4
```

```
Destructor called.3
```

例 2.25 中声明了类 CBox，类中有一个构造函数和一个析构函数，当构造函数运行时会输出一句话，方便判断构造函数执行的时机；同样的，当析构函数运行时也会输出一句话，方便判断析构函数执行的时机。

在一般情况下，调用析构函数的次序正好与调用构造函数的次序相反：最先被调用的构造函数，其对应的（同一对象中的）析构函数最后被调用；而最后被调用的构造函数，其对应的析构函数最先被调用。如上所示，先执行 box2 的析构函数，再执行 box1 的析构函数。可以简记为：先构造的后析构，后构造的先析构，它相当于一个栈，先进后出。但是，并不是在任何情况下都是按这一原则处理的。对象可以在不同的作用域中定义，可以有不同的存储类别，这些都会影响调用构造函数和析构函数的时机。下面归纳一下什么时候调用构造函数和析构函数。

(1) 在全局范围中定义的对象（即在所有函数之外定义的对象），它的构造函数在文件中的所有函数（包括 main 函数）执行之前调用。但如果一个程序中有多文件，而不同的文件中都定义了全局对象，则这些对象的构造函数的执行顺序是不确定的。当 main 函数执行完毕或调用 exit 函数时（此时程序终止），调用析构函数。

(2) 如果定义的是局部自动对象（如在函数中定义对象），则在建立对象时调用其构造函数。如果函数被多次调用，则在每次建立对象时都要调用构造函数。在函数调用结束、对象释放时先调用析构函数。

(3) 如果在函数中定义静态（static）局部对象，则只在程序第一次调用此函数建立对象时调用构造函数一次，在调用结束时对象并不释放，因此也不调用析构函数，只在 main 函数结束或调用 exit 函数结束程序时，才调用析构函数。例如，在一个函数中定义了以下两个对象：

```
void func(){
    Box box1;           // 定义自动局部对象
    static Box box2;     // 定义静态局部对象
}
```

在调用 func 函数时，先调用 box1 的构造函数，再调用 box2 的构造函数。在 func 调用结束时，box1 是要释放的（因为它是自动局部对象），因此要调用 box1 的析构函数。而 box2 是静态局部对象，在 func 调用结束时并不需要释放，因此不需要调用 box2 的析构函数，直到程序结束释放 box2 时，才调用 box2 的析构函数。由此可以看到，box2 是后调用构造函数的，但并不先调用其析构函数，原因是两个对象的存储类别、生命周期都不同。

2.2 继承与派生

1. 继承与派生的一般形式

继承与派生在 C++ 中也是经常使用的，比如设计一个箱子类可以用以下代码实现：

```
Class CBox{
```

```

public:
    int volume(){
        return height*width*length;
    }
    void display(){
        cout<<height<<endl;
        cout<<width<<endl;
        cout<<length<<endl;
    }
private:
    int height,width,length;
};

```

但假设现在有一批箱子比较特殊，有不同的颜色，且重量也都不一样，此时可以重新声明一个新的箱子的类，如下所示：

```

Class CBox_new{
public:
    int volume(){
        return height*width*length;
    }
    void display(){
        cout<<height<<endl;
        cout<<width<<endl;
        cout<<length<<endl;
        cout<<color<<endl;
        cout<<weight<<endl;
    }
private:
    int height,width,length;
    int color,weight;
};

```

可以看到上面的程序中有相当一部分是原来就已经有的，其实可以利用原来的 CBox 类作为基础，再加上新的内容即可，如下所示：

```

Class CBox_new::public CBox{           // 类 CBox_new 继承于类 CBox
public:
    void display1(){                   // 新增加的成员函数
        cout<<color<<endl;
        cout<<weight<<endl;
    }
private:
    int color,weight;                  // 新增加的成员变量
};

```

这里，Box_new 就是一个派生类。可见，声明派生类的一般形式为：

```

class 派生类名 : [继承方式] 基类名 {
    派生类新增加的成员
};

```

其中的继承方式包括 `public` (公用的)、`private` (私有的) 和 `protected` (受保护的), 此项是可选的, 如果不写此项, 则默认为 `private` (私有的)。

派生类里有两大部分内容: 从基类继承而来的和在声明派生类时增加的部分。派生类中接受了基类的全部内容, 这样可能出现有些基类的成员, 在派生类中是用不到的, 但是也必须继承过来的情况。这就会造成数据的冗余, 尤其多次派生后, 会在许多派生类对象中存在大量无用的数据, 不仅浪费了大量的空间, 而且在对象的建立、赋值、复制和参数的传递中, 花费许多无谓的时间, 从而降低了效率。因此, 实际开发中要根据派生类的需要慎重选择基类, 使冗余量最小。

2. 派生类的访问属性

(1) 派生类中包含基类成员和派生类自己增加的成员, 就产生了这两部分成员的关系和访问属性的问题。在建立派生类的时候, 并不是简单地把基类的私有成员直接作为派生类的私有成员, 把基类的公用成员直接作为派生类的公用成员; 实际上, 对基类成员和派生类自己增加的成员是按不同的原则处理的。具体来说, 在讨论访问属性时, 要考虑以下几种情况。

1) 基类的成员函数只能访问基类的成员, 而不能访问派生类的成员。

2) 派生类的成员函数可以访问基类的成员, 具体见后面详细描述; 派生类的成员函数也可以访问派生类成员。

3) 在派生类外可以访问基类的成员, 具体见后面详细描述; 在派生类外也可以访问派生类的公用成员, 而不能访问派生类的私有成员。

(2) 派生类的成员函数访问基类的成员和在派生类外访问基类的成员涉及如何确定基类的成员在派生类中的访问属性的问题, 不仅要考虑对基类成员所声明的访问属性, 还要考虑派生类所声明的对基类的继承方式, 根据这两个因素共同决定基类成员在派生类中的访问属性。在派生类中, 对基类的继承方式可以有 `public` (公用的)、`private` (私有的) 和 `protected` (保护的) 3 种。不同的继承方式决定了基类成员在派生类中的访问属性。简单地说可以总结为以下几点。

1) 公用继承 (`public inheritance`): 基类的公用成员和保护成员在派生类中保持原有访问属性, 其私有成员仍为基类私有。

2) 私有继承 (`private inheritance`): 基类的公用成员和保护成员在派生类中成了私有成员, 其私有成员仍为基类私有。

3) 受保护的继承 (`protected inheritance`): 基类的公用成员和保护成员在派生类中成了保护成员, 其私有成员仍为基类私有。保护成员的意思是, 不能被外界引用, 但可以被派生类的成员引用。

在多级派生的情况下, 各成员的访问属性仍按以上原则确定。假设类 A 为基类, 类 B 是类 A 的派生类, 类 C 是类 B 的派生类, 则类 C 也是类 A 的派生类; 类 B 称为类 A 的直接派生类, 类 C 称为类 A 的间接派生类; 类 A 是类 B 的直接基类, 是类 C 的间接基类。派生关系如图 2-3 所示。

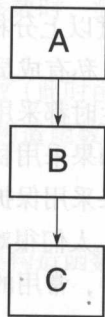


图 2-3 类 A、类 B 和类 C 的派生关系

如果声明了以下的类：

```
class A{
public:
    int var_A_pub;
protected:
    void func_A_pro();
    int var_A_pro;
private:
    int var_A_pri;
};
class B:public A{
public:
    void func_B_pub();
protected:
    void func_B_pro();
private:
    int var_B_pri;
};
class C:protected B{
public:
    void func_C_pub();
private:
    int var_C_pri;
};
```

则类 A 是类 B 的公用基类，类 B 是类 C 的保护基类。各成员在不同类中的访问属性如表 2-1 所示。

表 2-1 派生的各个成员的访问属性

	var_A_ pub	func_A_ pro	var_A_ pro	var_A_ pri	func_B_ pub	func_B_ pro	var_B_ pri	func_C_ pub	var_C_ pri
基类 A	公用	保护	保护	私有					
公用派生类 B	公用	保护	保护	不可访问	公用	保护	私有		
保护派生类 C	保护	保护	保护	不可访问	保护	保护	不可访问	公用	私有

通过以上分析，可以看到：无论哪一种继承方式，在派生类中是不能访问基类的私有成员的，私有成员只能被本类的成员函数所访问，毕竟派生类与基类不是同一个类。如果在多级派生时都采用公用继承方式，那么直到最后一级派生类都能访问基类的公用成员和保护成员。如果采用私有继承方式，经过若干次派生之后，基类的所有成员就会变成不可访问的了。如果采用保护继承方式，在派生类外是无法访问派生类中的任何成员的；而且经过多次派生后，人们很难清楚地记住哪些成员可以访问，哪些成员不能访问，很容易出错。因此，在实际中，常用的是公用继承。

3. 派生类的构造函数与析构函数

派生类的数据成员由所有基类的数据成员与派生类新增的数据成员共同组成，如果派生

类新增成员中包括其他类的对象(子对象), 派生类的数据成员中实际上还间接地包括了这些对象的数据成员。因此, 构造派生类的对象时, 必须对基类数据成员、新增数据成员和成员对象的数据成员进行初始化。派生类的构造函数必须要以合适的初值作为参数, 隐含调用基类和新增对象成员的构造函数, 来初始化它们各自的数据成员, 然后再加入新的语句对新增普通数据成员进行初始化。

派生类构造函数的一般格式如下:

```
<派生类名>::<派生类名>(<参数表>) : <基类名1>(<参数表1>),
.....,
<基类名n>(<参数表n>),
<子对象名1>(<参数表n+1>),
.....,
<子对象名m>(<参数表n+m>) {
    <派生类构造函数体>                // 派生类新增成员的初始化
}
```

对派生类的构造函数有以下几点说明:

(1) 对基类成员和子对象成员的初始化必须在成员初始化列表中进行, 新增成员的初始化既可以在成员初始化列表中进行, 也可以在构造函数体中进行。

(2) 派生类构造函数必须对这3类成员进行初始化, 其执行顺序是这样的: ①先调用基类构造函数; ②再调用子对象的构造函数; ③最后调用派生类的构造函数体。

(3) 当派生类有多个基类时, 处于同一层次的各个基类的构造函数的调用顺序取决于定义派生类时声明的顺序(自左向右), 而与在派生类构造函数的成员初始化列表中给出的顺序无关。

(4) 如果派生类的基类也是一个派生类, 则每个派生类只需负责其直接基类的构造, 依次上溯。

(5) 当派生类中有多个子对象时, 各个子对象构造函数的调用顺序也取决于在派生类中定义的顺序(自前至后), 而与在派生类构造函数的成员初始化列表中给出的顺序无关。

(6) 派生类构造函数提供了将参数传递给基类构造函数的途径, 以保证在基类进行初始化时能够获得必要的参数。因此, 如果基类的构造函数定义了一个或多个参数时, 派生类必须定义构造函数。

(7) 如果基类中定义了默认构造函数或根本没有定义任何一个构造函数(此时由编译器自动生成默认构造函数)时, 在派生类构造函数的定义中可以省略对基类构造函数的调用, 即省略"<基类名>(<参数表>)"这个语句。

(8) 子对象的情况与基类相同。

(9) 当所有的基类和子对象的构造函数都可以省略时, 可以省略派生类构造函数的成员初始化列表。

(10) 如果所有的基类和子对象构造函数都不需要参数, 派生类也不需要参数时, 派生类构造函数可以不定义。派生类构造函数的使用可以参考例2.26的程序。

【例 2.26】 派生类构造函数的使用举例。

```

#include<iostream>
#include<string>
using namespace std;
class CStudent{                                // 声明基类 Student
public:
    CStudent(int n,string nam,char s){        // 基类构造函数
        num=n;
        name=nam;
        sex=s;
    }
    ~CStudent(){                               // 基类析构函数
protected:
    int num;
    string name;
    char sex ;
};
class CStudent1: public CStudent{             // 声明派生类 Student1
public :                                       // 派生类的公用部分
    CStudent1 (int n,string nam,char s,int a,string ad): CStudent (n,nam,s){
                                                // 派生类构造函数
                                                // 在函数体中只对派生类新增的数据成员初始化
        age=a;
        addr=ad;
    }
    void show(){
        cout<<"num: "<<num<<endl;
        cout<<"name: "<<name<<endl;
        cout<<"sex: "<<sex<<endl;
        cout<<"age: "<<age<<endl;
        cout<<"address: "<<addr<<endl<<endl;
    }
    ~CStudent1(){                             // 派生类析构函数
private :                                    // 派生类的私有部分
    int age;
    string addr;
};
int main(){
    CStudent1 stud1(10010,"Wang-li",'f',19,"115 Beijing Road,Shanghai");
    CStudent1 stud2(10011,"Zhang-fun",'m',21,"213 Shanghai Road,Beijing");
    stud1.show();                             // 输出第一个学生的数据
    stud2.show();                             // 输出第二个学生的数据
    return 0;
}

```

程序的执行结果是:

```

num: 10010
name: Wang-li
sex: f
age: 19

```

```
address: 115 Beijing Road, Shanghai
```

```
num: 10011
```

```
name: Zhang-fun
```

```
sex: m
```

```
age: 21
```

```
address: 213 Shanghai Road, Beijing
```

例 2.26 中定义了类 CStudent、类 CStudent1，其中类 CStudent1 继承了类 CStudent。类 CStudent1 比基类新增了两个数据成员。基类 CStudent 有自己的带参构造函数，而派生类的构造函数，则只须对派生类新增的数据成员初始化，不过需要把基类的参数也给带进去。

析构函数的作用是在对象撤销之前，进行必要的清理工作。当对象被删除时，系统会自动调用析构函数。

析构函数比构造函数简单，没有类型，也没有参数。在派生时，派生类是不能继承基类的析构函数的，也需要通过派生类的析构函数去调用基类的析构函数。在派生类中可以根据需要定义自己的析构函数，用来对派生类中所增加的成员进行清理工作；基类的清理工作仍然由基类的析构函数负责。在执行派生类的析构函数时，系统会自动调用基类的析构函数和子对象的析构函数，对基类和子对象进行清理。

4. 派生类的构造函数与析构函数的调用顺序

前面已经提到，构造函数和析构函数的调用顺序是先构造的后析构，后构造的先析构。那么基类和派生类中的构造函数和析构函数的调用顺序是否也是如此呢？

构造函数的调用顺序规则如下所述。

1) 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。

2) 成员类对象构造函数。如果有多个成员类对象，则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。

3) 派生类构造函数。

而析构函数的调用顺序与构造函数的调用顺序正好相反，将上面 3 点内容中的顺序反过来用就可以了，即：首先调用派生类的析构函数；其次再调用成员类对象的析构函数；最后调用基类的析构函数。析构函数在下面 3 种情况时被调用。

1) 对象生命周期结束被销毁时（一般类成员的指针变量与引用都不自动调用析构函数）。

2) delete 指向对象的指针时，或 delete 指向对象的基类类型指针，而其基类虚函数是虚函数时。

3) 对象 i 是对象 o 的成员，o 的析构函数被调用时，对象 i 的析构函数也被调用。

下面用例 2.27 来说明构造函数的调用顺序。

【例 2.27】 构造函数的调用顺序。

```
#include<iostream>
using namespace std;
```

```

class CBase{
public:
    CBase () { std::cout<<"CBase::CBase()"<<std::endl; }
    ~ CBase () { std::cout<<"CBase::~~CBase()"<<std::endl; }
};

class CBase1:public CBase {
public:
    CBase1 () { std::cout<<"CBase::Base1()"<<std::endl; }
    ~ CBase1 () { std::cout<<"CBase::~~Base1()"<<std::endl; }
};

class CDerive{
public:
    CDerive () { std::cout<<"CDerive::CDerive()"<<std::endl; }
    ~ CDerive () { std::cout<<"CDerive::~~CDerive()"<<std::endl; }
};

class CDerive1:public CBase1{
private:
    CDerive m_derive;
public:
    CDerive1() { std::cout<<"CDerive1::CDerive1()"<<std::endl; }
    ~CDerive1() { std::cout<<"CDerive1::~~CDerive1()"<<std::endl; }
};

int main(){
    CDerive1 derive;
    return 0;
}

```

程序的执行结果是：

```

CBase::CBase()
CBase::Base1()
CDerive::CDerive()
CDerive1::CDerive1()
CDerive1::~~CDerive1()
CDerive::~~CDerive()
CBase::~~Base1()
CBase::~~CBase()

```

例 2.27 中声明了 4 个类，CBase1 继承于 CBase、CDerive1 继承于 CBase1、CDerive1 中有 CDerive 的成员变量，最后定义一个 CDerive1 对象，用来确定各种基类、派生类的构造函数和析构函数的执行顺序。执行结果与上文中描述的调用顺序相符。

总的来说，构造函数的调用顺序是：①如果存在基类，那么先调用基类的构造函数，如果基类的构造函数中仍然存在基类，那么程序会继续进行向上查找，直到找到它最早的基类进行初始化（如上例中类 Derive1，继承于类 Base 与 Base1）；②如果所调用的类中定义的时候存在对象被声明，那么在基类的构造函数调用完成以后，再调用对象的构造函数（如上例

在类 `Derive1` 中声明的对象 `Derive m_derive`); ③调用派生类的构造函数 (如上例最后调用的是 `Derive1` 类的构造函数)。

2.3 类的多态

1. 多态

多态, 顾名思义, 是一个事物有多种形态的意思。在 C++ 程序设计中, 多态性是指具有不同功能的函数可以用同一个函数名, 这样就可以用一个函数名调用不同内容的函数。在面向对象方法中一般是这样表述多态性的: 向不同的对象发送同一个消息, 不同的对象在接收时会产生不同的行为 (即方法); 也就是说, 每个对象可以用自己的方式去响应共同的消息。所谓消息, 就是调用函数, 不同的行为就是指不同的实现, 即执行不同的函数。

【例 2.28】基类和派生类成员函数同名覆盖时的执行选择。

```
#include<iostream>
using namespace std;
class A{
public:
    A(){}
    virtual void foo(){
        cout<<"This is A."<<endl;
    }
};

class B : public A{
public:
    B(){}
    void foo(){
        cout<<"This is B."<<endl;
    }
};

int main(){
    A a;
    a.foo();
    B b;
    b.foo();
    return 0;
}
```

程序的执行结果:

```
This is A.
This is B.
```

例 2.28 中声明了两个类 (类 A 和类 B), 注意类 A 中的 `foo` 函数和类 B 中的 `foo` 函数不是重载函数, 它们不仅函数名相同, 而且函数类型和参数个数都相同, 但两个同名函数不在

同一个类中，而是分别在基类和派生类中，属于同名覆盖。若是重载函数，二者的参数个数和参数类型必须至少有一者不同，否则系统无法确定调用哪一个函数。而此处定义了一个 A 类的对象 a 和 B 类的对象 b，有所区别，所以会分别执行各个类的 foo 函数。

人们提出这样的设想，能否用同一个调用形式，既能调用派生类又能调用基类的同名函数。在程序中不是通过不同的对象名去调用不同派生层次中的同名函数，而是通过指针调用它们。C++ 中的虚函数就是用来解决这个问题的。虚函数的作用是允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数。

再看下面的例 2.29，基类和派生类中有同名的函数 display，就是使用虚函数，使得基类指针可以访问派生类中的同名函数。

【例 2.29】 使用虚函数可以使得基类指针访问派生类中的同名函数。

```
#include <iostream>
#include <string>
using namespace std;
/* 声明基类 Box*/
class Box{
public:
    Box(int,int,int);           // 声明构造函数
    virtual void display();     // 声明输出函数
protected:
    int length,height,width;   // 受保护成员，派生类可以访问
};
/*Box 类成员函数的实现*/
Box::Box (int l,int h,int w){   // 定义构造函数
    length =l;
    height =h;
    width =w;
}
void Box::display(){           // 定义输出函数
    cout<<"length:" << length <<endl;
    cout<<"height:" << height <<endl;
    cout<<"width:" << width <<endl;
}
/* 声明公用派生类 FilledBox*/
class FilledBox : public Box{
public:
    FilledBox (int, int, int, int, string); // 声明构造函数
    virtual void display();                // 虚函数
private:
    int weight;                            // 重量
    string fruit;                          // 装着的水果
};
/* FilledBox 类成员函数的实现*/
void FilledBox :: display(){              // 定义输出函数
    cout<<"length:"<< length <<endl;
    cout<<"height:"<< height <<endl;
    cout<<"width:"<< width <<endl;
    cout<<"weight:"<< weight <<endl;
```

```

    cout<<"fruit:"<< fruit <<endl;
}
FilledBox:: FilledBox (int l, int h, int w, int we, string f ) : Box(l,h,w), weight(we),
fruit(f){}
int main(){
    Box box(1,2,3);
    FilledBox fbox(2,3,4,5,"apple");
    Box *pt = &box;
    pt->display( );
    pt = &fbox;
    pt->display( );
    return 0;
}

```

// 主函数
// 定义 Student 类对象 stud1
// 定义 FilledBox 类对象 fbox
// 定义指向基类对象的指针变量 pt

程序的执行结果是:

```

length:1
height:2
width:3
length:2
height:3
width:4
weight:5
fruit:apple

```

例 2.25 中声明了一个类 Box 和一个继承于类 Box 的类 FilledBox。类 Box 中有一个成员函数 display，类 FilledBox 中也有一个成员函数 display，现在将基类 Box 中的成员函数 display 定义为虚函数，就能使得基类对象的指针变量既可以访问基类的成员函数 display，也可以访问派生类的成员函数 display。

例 2.25 就展现了虚函数的奇妙作用。现在用同一个指针变量（指向基类对象的指针变量），不但输出了 box 的全部数据，而且还输出了 fbox 的全部数据，这就说明已调用了 fbox 的 display 函数。这表明用同一种调用形式 "pt->display()", 而且 pt 是同一个基类指针，也可以调用同一类族中不同类的虚函数。这就是多态性，即对同一消息，不同对象有不同的响应方式。

说明：本来基类指针是用来指向基类对象的，如果用它指向派生类对象，则需要进行指针类型转换，即将派生类对象的指针先转换为基类的指针，所以基类指针指向的是派生类对象中的基类部分。如果基类中的 display 函数不是虚函数，是无法通过基类指针去调用派生类对象中的成员函数的。虚函数突破了这一限制，在派生类的基类部分中，派生类的虚函数取代了基类原来的虚函数，因此在使基类指针指向派生类对象后，调用虚函数时就调用了派生类的虚函数。要注意的是，只有用 virtual 声明了虚函数后才具有以上作用，如果不声明为虚函数，企图通过基类指针调用派生类的非虚函数则是不行的。

当把基类的某个成员函数声明为虚函数后，就允许在其派生类中对该函数重新定义，赋予它新的功能，并且可以通过指向基类的指针指向同一类族中不同类的对象，从而调用其中的同名函数。虚函数实现了同一类族中不同类的对象可以对同一函数调用作出不同的响应的动态多态性。

虚函数的使用方法如下所述。

(1) 在基类用 `virtual` 关键字声明成员函数为虚函数。

这样就可以在派生类中重新定义此函数，为它赋予新的功能，并能方便地被调用。在类外定义虚函数时，不必再加 `virtual` 关键字。

(2) 在派生类中重新定义此函数，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同，并根据派生类的需要重新定义函数体。

C++ 规定，当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数。因此在派生类重新声明该虚函数时，可以加 `virtual` 关键字，也可以不加，但一般习惯在每一层声明该函数时都加 `virtual` 关键字，使程序更加清晰。如果在派生类中没有对基类的虚函数重新定义，则派生类简单地继承其直接基类的虚函数。

(3) 定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象。

通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。

(4) 通过虚函数与指向基类对象的指针变量的配合使用，就能方便地调用同一类族中不同类的同名函数，只要先用基类指针指向即可。如果指针不断地指向同一类族中不同类的对象，就能不断地调用这些对象中的同名函数。正如前面所说，不断地告诉出租车司机要去的目的地，然后司机把你送到你要去的地方。

需要说明以下几点：①有时在基类中定义的非虚函数会在派生类中被重新定义，如果用基类指针调用该成员函数，则系统会调用对象中基类部分的成员函数；②如果用派生类指针调用该成员函数，则系统会调用派生类对象中的成员函数，这并不是多态性行为（使用的是不同类型的指针），没有用到虚函数的功能。

2. 虚函数的使用

(1) 使用虚函数时，有两点要注意，如下所述。

1) 只能用 `virtual` 关键字声明类的成员函数，使它成为虚函数，而不能将类外的普通函数声明为虚函数。因为虚函数的作用是允许在派生类中对基类的虚函数重新定义。显然，它只能用于类的继承层次结构中。

2) 一个成员函数被声明为虚函数后，在同一类族中的类就不能再定义一个非 `virtual` 的但与该虚函数具有相同的参数（包括个数和类型）和函数返回值类型的同名函数。

(2) 根据什么考虑是否把一个成员函数声明为虚函数呢？主要考虑以下几点。

1) 首先看成员函数所在的类是否会作为基类。然后看成员函数在类的继承后有无可能被更改功能，如果希望更改其功能，一般应该将它声明为虚函数。

2) 如果成员函数在类被继承后的功能不需要被修改，或派生类用不到该函数，则不要把它声明为虚函数。不要仅仅考虑到要作为基类而把类中的所有成员函数都声明为虚函数。

3) 应考虑对成员函数的调用是通过对象名还是通过基类指针或引用去访问，如果是通过基类指针或引用去访问的，则应当声明为虚函数。

4) 有时，在定义虚函数时并不定义其函数体，即函数体是空的。它的作用只是定义了一个虚函数名，具体功能留给派生类去添加。

需要说明的是：使用虚函数，系统要有一定的空间开销。当一个类带有虚函数时，编译系统会为该类构造一个虚函数表，它是一个指针数组，用于存放每个虚函数的入口地址。系统在进行动态关联时的时间开销是很少的，因此，多态是高效的。

3. 纯虚函数

纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”，如下所示：

```
virtual void fuction()=0;
```

而虚函数的定义是：

```
virtual void fuction();
```

为了方便使用多态特性，常常需要在基类中定义虚函数。但在很多情况下，用基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但用动物本身生成对象明显不合常理。为了解决上述问题，从而引入了纯虚函数的概念，将函数定义为纯虚函数，则编译器要求在派生类中必须予以重载以实现多态性。同时含有纯虚函数的类称为抽象类，它不能生成对象。如果一个类中含有纯虚函数，那么任何试图对该类进行实例化的语句都是错误的，因为抽象基类是不能被直接调用的，而必须被子类继承重载以后，再根据要求调用其子类的方法，且在子类中一定要实现纯虚函数的定义，不然编译时会出错。

【例 2.30】 纯虚函数的使用举例。

```
class Animail{
public:
    virtual void GetColor() = 0;
};
class Dog : public Animail{
public:
    virtual void GetColor() {cout <<"Yellow"endl;};
};
class Pig : public Animail{
public:
    virtual void GetColor() {cout <<"White"<<endl;};
};
int main(){
    Animail cAnimail;
    return 0;
}
```

该程序编译失败，因为在例 2.30 中声明了一个动物类 (Animail)，类中有一函数 GetColor 可取得动物颜色，但动物有很多很多种，颜色自然无法确定，所以就把它声明为纯虚函数，也就是只声明函数名不去定义 (实现) 它，不能通过编译。有一点需要注意，纯虚函数不能实例化，但可以声明指针，所以上面的程序编译时，编译器会告诉你：由于它的原因，无法抽象类 Animail，并且警告你 GetColor() 没有定义，所以报错。

4. 析构函数

在 C++ 中，构造函数不能声明时为虚函数，这是因为编译器在构造对象时，必须知道确切类型，才能正确地生成对象；其次，在构造函数执行之前，对象并不存在，无法使用指向此对象的指针来调用构造函数。然而，析构函数可以声明为虚函数；C++ 明确指出，当 derived class 对象经由一个 base class 指针被删除、而该 base class 带着一个 non-virtual 析构函数，会导致对象的 derived 成分没被销毁掉，如例 2.31 所示。

【例 2.31】析构函数不是虚函数容易引发内存泄漏。

```
#include<iostream>
using namespace std;
class Base{
public:
    Base(){ std::cout<<"Base::Base()"<<std::endl; }
    ~Base(){ std::cout<<"Base::~~Base()"<<std::endl; }
};
class Derive:public Base{
public:
    Derive(){ std::cout<<"Derive::Derive()"<<std::endl; }
    ~Derive(){ std::cout<<"Derive::~~Derive()"<<std::endl; }
};
int main(){
    Base* pBase = new Derive();
    /* 这种 base classed 的设计目的是为了用来“通过 base class 接口处理 derived class 对象”*/
    delete pBase;
    return 0;
}
```

程序的执行结果是：

```
Base::Base()
Derive::Derive()
Base::~~Base()
```

例 2.31 中声明了两个类 Base 和类 Derive，类 Derive 继承于类 Base，两个类各自有构造函数和析构函数，并且基类和派生类的析构函数都是非虚函数。从上面的执行结果可以看出，析构函数的调用结果是存在问题的，也就是说析构函数只做了局部销毁工作，这可能形成资源泄漏、损坏数据结构等问题。而解决此问题的方法很简单，只要给基类一个 virtual 析构函数即可，如例 2.32 所示。

【例 2.32】基类的析构函数为虚函数。

```
#include<iostream>
using namespace std;
class Base{
public:
    Base(){ std::cout<<"Base::Base()"<<std::endl; }
    virtual ~Base(){ std::cout<<"Base::~~Base()"<<std::endl; }
};
```

```

class Derive:public Base{
public:
    Derive(){ std::cout<<"Derive::Derive()"<<std::endl; }
    ~Derive(){ std::cout<<"Derive::~~Derive()"<<std::endl; }
};

int main(){
    Base* pBase = new Derive();
    delete pBase;
    return 0;
}

```

输出结果是:

```

Base::Base()
Derive::Derive()
Derive::~~Derive()
Base::~~Base()

```

例 2.32 与例 2.31 的区别只在于是否把基类的析构函数声明为虚函数。例 2.32 中派生类和基类都能正常析构了, 这样的结果正是我们所希望的。虚函数是多态的基础, 在 C++ 中没有虚函数就无法实现多态特性; 因为不声明为虚函数就不能实现“动态联编”, 就不能实现多态。

5. 单例模式

要理解单例模式, 只需要一个实例就可以了。比如, 一台计算机上可以连好几台打印机, 但是这个计算机上的打印程序只能有一个, 这里就可以通过单例模式来避免两个打印作业同时输出到打印机中, 即在整个的打印过程中只有一个打印程序的实例。对于这种问题, 《设计模式》一书中给出了一种很不错的实现, 定义一个单例类, 使用类的私有静态指针变量指向类的唯一实例, 并用一个公有的静态方法来获取该实例。单例模式的作用就是保证在整个应用程序的生命周期中的任何一个时刻, 单例类的实例都只存在一个 (当然也可以不存在)。

单例模式通过类本身来管理其唯一实例, 唯一的实例是类的一个普通对象, 但设计这个类时, 让它只能创建一个实例并提供对此实例的全局访问, 具体可以参见下面的例 2.33。

【例 2.33】单例模式使用举例。

```

#include<iostream>
using namespace std;
class CSingleton{
private:
    CSingleton(){ // 构造函数是私有的
    }
    static CSingleton *m_pInstance;
public:
    static CSingleton * GetInstance(){
        if(m_pInstance == NULL) // 判断是否第一次调用
            m_pInstance = new CSingleton();
        return m_pInstance;
    }
}

```

```

};
CSingleton * CSingleton::m_pInstance=NULL;           // 初始化静态数据成员
int main(){
    CSingleton *s1= CSingleton::GetInstance();
    CSingleton *s2= CSingleton::GetInstance();
    if(s1==s2){
        cout<<"s1=s2"<<endl;           // 程序的执行结果是输出了 s1=s2
    }
    return 0;
}

```

程序的执行结果是：

```
s1=s2
```

例 2.33 中，用户访问实例的唯一方法只有 `GetInstance()` 成员函数。如果不通过这个函数，任何创建实例的尝试都将失败，因为类的构造函数是私有的。`GetInstance()` 的返回值是当这个函数首次被访问时被创建的，所有 `GetInstance()` 之后的调用都返回相同实例的指针。

单例类 `CSingleton` 有以下特征：①有一个指向唯一实例的静态指针 `m_pInstance`，并且是私有的；②有一个公有的函数，可以获取这个唯一的实例，并且在需要的时候创建该实例；③其构造函数是私有的，这样就不能从别处创建该类的实例。

2.4 本章小结

本章从对象的封装、继承、多态这三大特征来带领读者学习面向对象的 C++。这 3 个特征分别解决了以下问题。

(1) 封装：找到变化并且把它封装起来，就可以在不影响其他部分的前提下修改或扩展被封装的变化部分。封装解决了程序的可扩展性。

(2) 继承：子类继承父类，可以继承父类的方法及属性，实现了多态以及代码的重用，解决了系统的重用性和扩展性，但是继承破坏了封装，因为其对子类开放的，修改父类会导致所有子类的改变，因此继承一定程度上又破坏了系统的可扩展性，所以继承需要慎用。继承是在程序开发过程中重构得到的，而不是程序设计之初就使用继承，很多面向对象开发者滥用继承，结果可能造成后期的代码解决不了需求的变化。因此优先使用组合，而不是继承，是面向对象开发中一个重要的经验。

(3) 多态：接口的多种不同的实现方式即为多态。接口是对行为的抽象，上面在“封装”中提到，找到变化部分并封装起来，但是封装起来后，怎么适应接下来的变化？这正是接口的作用，接口的主要目的是为不相关的类提供通用的处理服务，从而实现系统的可维护性、可扩展性。

因此，面向对象实现了人们追求的系统可维护性、可扩展性、可重用性。

第 3 章将学习 STL，了解如何更高效地写程序，玩转 C++。



第3章 Chapter 3

常用 STL 的使用

3.1 STL 是什么

当今时代是一个信息时代，科技的发展所带来的便利影响了人们生活中的每个细节，STL 就是这个时代组件化大生产的产物。正如其他科技成果一样，C++ 程序员也应该努力使自己适应并充分利用这个“高科技成果”。

STL 是一个标准模板库，是一个高效的 C++ 程序库。接下来将以一个实例程序为例，逐步介绍 STL 的内容与功能。

假设需要从标准输入（一般是键盘）读入一些整型数据，再对它们排序，然后将结果输出到标准输出设备（一般是显示器屏幕），那程序可以如例 3.1 所示。

【例 3.1】 用原始的方式将一些数据进行排序。

```
#include<iostream>
#include<stdlib.h>
using namespace std;
#define max_size 10
// 比较两个数的大小
// 如果比较函数返回大于 0，qsort 就认为 a>b
// 如果比较函数返回等于 0，qsort 就认为 a=b
// 如果比较函数返回小于 0，qsort 就认为 a<b
int cmp(const void *a,const void *b){
    return *(int *)a-*(int *)b;
}
int main(){
    int arr[max_size];
    int n=0;
    // 从标准输入设备中读入整数，同时累计输入个数，直到输入的是非整型数据为止
```



```

    for(;;n++){
        cin>>arr[n];
    }
    qsort(arr,n,sizeof(int),cmp);
    for(int i=0;i<n;i++){
        cout<<arr[i]<<" ";
    }
    return 0;
}

```

以下是某次运行的结果：

输入：0 9 2 1 5

输出：0 1 2 5 9

例 3.1 中用了—个可以放 10 个整数的整型数组，来存放输入的数据，规定从标准输入设备中读入整数，同时累计输入个数，直到输入的是非整型数据为止；还用了 C 标准库的快速排序 `qsort` 函数来对输入的整数进行排序，并将排序结果输出到标准输出设备上。

然而，这个程序并不像看起来那么健壮（robust）。如果用户输入的数字数超过 `max_size` 所规定的上限，就会出现数组越界问题。为了弥补程序中的这一缺陷，必须采用下述方案中的一种。

（1）采用大容量的静态数据分配。

（2）限定输入的数据个数。

（3）采用动态内存分配。

第一种方案比较简单，所做的只是将 `max_size` 改大一点，比如 1000 或者 10 000。但是，严格讲这并不能最终解决问题，隐患仍然存在。此外，分配一个大数组，通常是在浪费空间，因为大多数情况下，数组中的一部分空间并没有被利用。

再来看看第二种方案，通过在第一个 `for` 循环中加入一个限定条件，可以使问题得到解决。比如：`for (int n = 0; cin >> num[n] && n < max_size; n ++);` 但是这个方案同样不甚理想，尽管不会使程序崩溃，但失去了灵活性，使用户无法输入更多的数。

看来只有选择第三种方案了，利用指针以及动态内存分配可以妥善解决上述问题，并且使程序具有良好的灵活性。这需要用到 `new`、`delete` 操作符，或者 `malloc()`、`realloc()` 和 `free()` 函数，但是为此，程序将牺牲其简洁性，使代码量陡增，代码的处理逻辑也不再像原先看起来那么清晰了。很难保证不会在处理这个问题的时候出错，很多程序的 bug 往往就是这样产生的。同时，`stdlib.h` 库提供了 `qsort` 函数，避免了自己实现排序算法的麻烦。总之，需要考虑的问题越来越多。

接下来就将从 STL 的角度分析并解决这类问题。

3.2 string

字符串处理问题是 C++ 语言编程中经常遇到的问题，熟练地掌握字符串处理的方法，可

以增强对字符串的存储和其处理的理解，从而写出高效的 C++ 程序。

在前面第 1 章讲到，字符串可以用字符指针 `char*`、字符数组等来表示，先来回顾一下字符指针和字符数组的使用注意点。

比如下面这几行代码：

```
char str[12] = "Hello";
char *p = str;
*p = 'h';           // 改变第一个字母
```

再看这几行代码：

```
char *ptr = "Hello";
*ptr = 'h';         // 错误
```

第一个字符串时用数组开辟的，它是可以改变的变量。而第二个字符串则是一个常量，也就是不可改变的值。`ptr` 只是指向它的指针而已，而不能改变指向的内容。这部分区别看两者的汇编语言即可明了：

`char p[] = "Hello";` 的汇编代码如下：

```
004114B8  mov     eax,dword ptr [string "Hello" (4166FCh)]
004114BD  mov     dword ptr [ebp-10h],eax
004114C0  mov     cx,word ptr ds:[416700h]
004114C7  mov     word ptr [ebp-0Ch],cx
```

`char *ptr = "Hello";` 的汇编代码如下：

```
004114CB  mov     dword ptr [ebp-1Ch],offset string "Hello" (4166FCh)
```

可见用数组和用指针是完全不相同的。

要想通过指针来改变常量是错误的，正确的写法应该用 `const` 指针，如下所示：

```
const char *ptr = "Hello";
```

除了以上限制外，字符数组、字符指针的字符串会有要考虑内存释放是否足够、字符串长度等问题，因此本章主要讲解 `string`。它是一个字符串的类，它集成的操作函数足以完成大多数情况下的需要。可以用“=”进行赋值操作，“==”进行比较，“+”做串联，使用非常简单，甚至可直接把它看作 C++ 的基本数据类型。

为了在程序中使用 `string` 类型，必须包含头文件 `<string>`，如下所示：

```
#include<string>
```



注意 `string.h` 和 `cstring` 都不是 `string` 类的头文件，这两个头文件主要定义 C 语言风格字符串操作的一些方法，譬如 `strlen()`、`strcpy()` 等。`string.h` 是 C 语言的头文件格式，而 `cstring` 是 C++ 风格的头文件，但是和 `<string.h>` 是一样的，它的目的是为了和 C 语言兼容。

1. string 类的实现

实现 string 类是一道考验 C++ 基础知识的好题，接下来先看这样的一道题目，了解 string 类的内部实现。

已知类 string 的原型代码如下所示，请编写类 string 的 7 个类。

```
class String{
public:
    String(const char *str = NULL);           // 普通构造函数
    String(const String &other);              // 拷贝构造函数
    ~ String();                               // 析构函数
    String & operator =(const String &other); // 赋值函数
    String & operator +(const String &other); // 字符串连接
    bool operator ==(const String &other);   // 判断相等
    int getLength();                          // 返回长度
private:
    char *m_data;                            // 私有变量保存字符串
};
```

从上述程序可以看到，string 类其实是一个对字符串指针有一系列操作动作的类，也就是说，string 类的底层是一个字符串指针。这一题的参考答案以及注意点如下所示。

(1) 普通构造函数。

```
String::String(const char *str){
    if(str==NULL){
        m_data = new char[1];
        *m_data = '\0';
        // 对空字符串自动申请存放结束标志 '\0' 的加分点：对 m_data 加 NULL 判断
    }
    else{
        int length = strlen(str);
        m_data = new char[length+1];
        strcpy(m_data, str);
    }
}
```

普通构造函数里需要注意的是，传入的是个 char* 类型的字符串。如果传入的 str 是个空的字符串，那这个 string 就也是一个空的字符串，直接用 \0 赋值。如果传入的 str 是非空字符串，私有变量 m_data 就需要预留 length+1 的长度，其中“+1”是用来放最后的 '\0' 的，因为 strlen 计算字符串长度时，没把 '\0' 算进去。

(2) String 的析构函数。

```
String::~~String(){
    if(m_data){
        delete[] m_data;           // 或 delete m_data;
        m_data=0;
    }
}
```

析构函数的主要功能主要是删除成员变量，需要先判断字符指针是否为空，如果不为空，再将其删除，并将其指向 NULL。

(3) 拷贝构造函数。

```
String::String(const String &other){    // 输入参数为 const 型
    if(!other.m_data){                // 对 m_data 加 NULL 判断
        m_data=0;
    }
    m_data = new char[strlen(other.m_data)+1];
    strcpy(m_data, other.m_data);
}
```

拷贝构造函数里需要注意的是，传入的参数是个常引用，这样可以不用新增一个栈变量和参数内容可以保持不变，不被修改。

(4) 赋值函数。

```
String & String::operator =(const String &other){
    // 输入参数为 const 型
    if(this != &other){                // 检查是否自赋值
        delete[] m_data;                // 释放原有的内存资源
        if(!other.m_data){              // 对 m_data 作 NULL 判断
            m_data=0;
        }
        else{
            m_data=new char[strlen(other.m_data)+1];
            strcpy(m_data, other.m_data);
        }
    }
    return *this;                        // 返回本对象的引用
}
```

赋值函数里需要注意的是，如果传入的参数内容已与本身的内容一致，则不需要赋值。如果传入的参数内容与本身内容不一致，需要先清空本身的内容。

(5) 字符串连接。

```
String & String::operator +(const String &other){
    String newString;
    if(!other.m_data){
        newString=*this;
    }
    else if(!m_data){
        newString=other;
    }
    else{
        newString.m_data=new char[strlen(m_data)+strlen(other.m_data)+1];
        strcpy(newString.m_data,m_data);
        strcat(newString.m_data,other.m_data);
    }
}
```

```
return newString;
}
```

字符串连接函数里需要注意的分3种情况：传入的参数内容为空、本身内容为空或两者内容都不为空。

(6) 判断相等。

```
bool String::operator==(const String &other){
    if(strlen(m_data)!=strlen(other.m_data)){
        return false;
    }
    else{
        return strcmp(m_data,other.m_data)?false:true;
    }
}
```

判断相等函数，返回值只有 true 和 false，先判断长度是否一致，再判断内容是否一致。

(7) 返回长度。

```
int String::getLength(){
    return strlen(m_data);
}
```

返回长度函数，只需用 strlen 直接计算 char* 的长度即可。

2. string 声明方式

字符串变量的声明形式如下：

```
string Str;
```

这样就声明了一个字符串变量，但既然是一个类，就有构造函数和析构函数。上面的声明没有传入参数，所以就直接使用了 string 的默认的构造函数，这个函数所做的就是把 Str 初始化为一个空字符串。string 类的构造函数和析构函数如下所示：

string s;	// 生成一个空字符串 s
string s(string str)	// 拷贝构造函数 生成 str 的复制品
string s(string str,int stridx)	// 将字符串 str 内“始于位置 stridx”的部分当作
	// 字符串的初值
string s(char *str,int stridx,int strlen)	// 将字符串 str 内“始于 stridx 且长度顶多 strlen”
	// 的部分作为字符串的初值
string s(char *cstr)	// 将 C 字符串作为 s 的初值
string s(char *chars,int chars_len)	// 将 C 字符串前 chars_len 个字符作为字符串 s 的
	// 初值
string s(int num,char c)	// 生成一个字符串，包含 num 个 c 字符
string s(char *beg,char *end)	// 以区间 beg;end(不包含 end) 内的字符作为字符
	// 串 s 的初值
s.~string()	// 销毁 s 字符，释放内存

string 类的声明如例 3.2 所示。

【例 3.2】 string 的声明。

```

#include<iostream>
#include<string>
using namespace std;
int main(){
    string str1="Spend all your time waiting.";
    string str2="For that second chance.";
    string str3(str1,6);                // "all your time waiting."
    string str4(str1,6,3);              // "all"
    char ch_music[]={"Sarah McLachlan"};
    string str5=ch_music;
    string str6(ch_music);
    string str7(ch_music,5);            // "Sarah"
    string str8(4,'a');                 // aaaa
    string str9(ch_music+6,ch_music+14); // " McLachlan"
    cout<<"str1:"<<str1<<endl;
    cout<<"str2:"<<str2<<endl;
    cout<<"str3:"<<str3<<endl;
    cout<<"str4:"<<str4<<endl;
    cout<<"str5:"<<str5<<endl;
    cout<<"str6:"<<str6<<endl;
    cout<<"str7:"<<str7<<endl;
    cout<<"str8:"<<str8<<endl;
    cout<<"str9:"<<str9<<endl;
    return 0;
}

```

程序的执行结果是：

```

str1:Spend all your time waiting.
str2:For that second chance.
str3:all your time waiting.
str4:all
str5:Sarah McLachlan
str6:Sarah McLachlan
str7:Sarah
str8:aaaa
str9:McLachla

```

例 3.2 中展示了声明一个 string 字符串的各种方式。

3. C++ 字符串和 C 字符串的转换

C++ 提供的由 C++ 字符串转换成对应的 C 字符串的方法是使用 `data()`、`c_str()` 和 `copy()` 来实现。其中，`data()` 以字符数组的形式返回字符串内容，但并不添加 '\0'；`c_str()` 返回一个以 '\0' 结尾的字符数组；而 `copy()` 则把字符串的内容复制或写入既有的 `c_string` 或字符数组内。需要注意的是，C++ 字符串并不以 '\0' 结尾。

`c_str()` 语句可以生成一个 `const char *` 指针，并指向空字符的数组。这个数组的数据是临时的，当有一个改变这些数据的成员函数被调用后，其中的数据就会失效。因此要么用现

转换，要么把它的数据复制到用户自己可以管理的内存中后再转换。

【例 3.3】c_str() 使用方法举例。

```
#include<iostream>
#include<string>
using namespace std;
int main(){
    string str="Hello world.";
    const char * cstr=str.c_str();
    cout<<cstr<<endl;
    str="Abcd.";
    cout<<cstr<<endl;
    return 0;
}
```

程序的执行结果是：

```
Hello world.
Abcd.
```

如例 3.3 所示，改变了 str 的内容，cstr 的内容也会随着改变。所以上面如果继续使用 C 指针的话，导致的错误将是不可想象的。既然 C 指针指向的内容容易失效，就可以考虑把数据复制出来解决问题。

【例 3.4】将 c_str() 里的内容复制出来以保持有效性。

```
#include<iostream>
#include<string>
#include<string.h>
using namespace std;
int main(){
    char * cstr=new char[20];
    string str="Hello world.";
    strcpy(cstr,str.c_str());
    cout<<cstr<<endl;
    str="Abcd.";
    cout<<cstr<<endl;
    return 0;
}
```

程序的执行结果：

```
Hello world.
Hello world.
```

例 3.4 中用 strcpy 函数将 str.c_str() 的内容复制到 cstr 里了，这样就能保证 cstr 里的内容不随着 str 的内容改变而改变了。

copy(p,n,size_type _Off = 0) 这句表明从 string 类型对象中至多复制 n 个字符到字符指针 p 指向的空间中，并且默认从首字符开始，也可以指定开始的位置（从 0 开始计数），返回真

正从对象中复制的字符。不过用户要确保 *p* 指向的空间足够来保存 *n* 个字符。

【例 3.5】 string.copy 用法的详细举例。

```
#include<iostream>
#include<string>
using namespace std;

int main (){
    size_t length;
    char buffer[8];
    string str("Test string.....");
    cout<<"str:"<<str<<endl;

    length=str.copy(buffer,7,5);
    buffer[length]='\0';
    cout<<"str.copy(buffer,7,5),buffer contains: "<<buffer<<endl;

    length=str.copy(buffer,str.size(),5);
    buffer[length]='\0';
    cout<<"str.copy(buffer,str.size(),5),buffer contains:"<<buffer<<endl;

    length=str.copy(buffer,7,0);
    buffer[length]='\0';
    cout<< "str.copy(buffer,7,0),buffer contains:"<<buffer<<endl;

    length=str.copy(buffer,7); // 缺省参数 pos, 默认 pos=0;
    buffer[length]='\0';
    cout<<"str.copy(buffer,7),buffer contains:"<<buffer<<endl;

    length=str.copy(buffer,string::npos,5);
    buffer[length]='\0';
    cout<<"string::npos:"<<(int)(string::npos)<<endl;
    cout<<"buffer[string::npos]:"<<buffer[string::npos]<<endl;
    cout<<"buffer[length-1]:"<<buffer[length-1]<<endl;
    cout<<"str.copy(buffer,string::npos,5),buffer contains:"<<buffer<<endl;

    length=str.copy(buffer,string::npos);
    buffer[length]='\0';
    cout<<"str.copy(buffer,string::npos),buffer contains:"<<buffer<<endl;
    cout<<"buffer[string::npos]:"<<buffer[string::npos]<<endl;
    cout<<"buffer[length-1]:"<<buffer[length-1]<<endl;
    return 0;
}
```

程序的执行结果是：

```
str:Test string.....
str.copy(buffer,7,5),buffer contains: string.
str.copy(buffer,str.size(),5),buffer contains:string.....
str.copy(buffer,7,0),buffer contains:Test st
```

```

str.copy(buffer,7),buffer contains:Test st
string::npos:-1
buffer[string::npos]:
buffer[length-1]:.
str.copy(buffer,string::npos,5),buffer contains:string.....
str.copy(buffer,string::npos),buffer contains:Test string.....
buffer[string::npos]:
buffer[length-1]:.

```

例 3.5 中展示了通过不同的方法用 `string` 的 `copy` 方法进行字符串的复制，常见的 api 的参数一般是把起始点信息放在前面，长度信息放在后面，如 `string` 的构造函数 `string s(char *str,int stridx,int strlen)`；而 `copy` 方法却是把长度放在起始点前面，这个是需要小心使用的。另外，`copy` 函数的第二个参数，除了可以是长度，也可以是一个位置，如 `string::npos`。

`string::npos` 是一个机器最大的正整数，不同机器不一样，如 64 位机器是 18446744073709551615，而 32 位机器则是 4294967295，类型是 `std::container_type::size_type`。可以将其强制转换成 `int`，就是 -1，这样就不会存在移植的问题。一般用 `npos` 表示 `string` 的结束位置。

`copy` 函数的第三个参数不填时则默认为 0，即从第一个字符开始。

综上，可以使用 `string` 的 `c_str()`、`data()`、`copy(p,n)`，从一个 `string` 类型得到一个 C 类型的字符数组。

4. `string` 和 `int` 类型的转换

(1) `int` 转 `string` 的方法。

这里需要用到 `snprintf()`，函数原型为：

```
int snprintf(char *str, size_t size, const char *format, ...)
```

它的功能主要是将可变个参数 (...) 按照 `format` 格式化字符串，然后将其复制到 `str` 中，具体如下所述。

1) 如果格式化后的字符串长度小于 `size`，则将此字符串全部复制到 `str` 中，并给其后添加一个字符串结束符 (`\0`)。

2) 如果格式化后的字符串长度不小于 `size`，则只将其中的 (`size-1`) 个字符复制到 `str` 中，并给其后添加一个字符串结束符 (`\0`)，返回值为欲写入的字符串长度。

3) 函数的返回值是若成功，则返回欲写入的字符串长度，若出错则返回负值。

4) 如果原始参数为 “...”，那么这是个可变参数。

【例 3.6】`snprintf` 的使用举例。

```

#include<stdio.h>
int main (){
    char a[20];
    int i = snprintf(a, 9, "%012d", 12345);
    printf("i = %d, a = %s", i, a);
    return 0;
}

```

程序的执行结果是：

```
i = 12, a = 00000001
```

例 3.6 中，%012d 的格式是指使输出的 int 型的数值以 12 位的固定位宽输出，如果不足 12 位，则在前面补 0；如果超过 12 位，则按实际位数输出。如果输出的数值不是 int 型，则进行强制类型转换为 int，之后按前面的格式输出。那就是先得到了 000000012345，再取前面 (9-1) 位，即 8 位，则是 00000001。

与此类似的，将 int 转换为 string，代码通常可以这么写：

```
static inline std::string i64tostr(long long a){
    char buf[32];
    snprintf(buf, sizeof(buf), "%lld", a);
    return std::string(buf);
}
```

(2) string 转 int 的方法。

这里需要用到 strtol, strtoll, strtoul 或 strtoull 函数，它们的函数原型分别如下所示：

```
long int strtol(const char *nptr, char **endptr, int base);
long long int strtoll(const char *nptr, char **endptr, int base);
unsigned long int strtoul(const char *nptr, char **endptr, int base);
unsigned long long int strtoull(const char *nptr, char **endptr, int base);
```

它们的功能是将参数 nptr 字符串根据参数 base 来转换成有符号的整型数、有符号的长整型数，无符号的整型数、无符号的长整型数。参数 base 范围从 2 ~ 36，或 0。参数 base 代表采用的进制方式，如 base 值为 10 则采用 10 进制；若 base 值为 16 则采用 16 进制数等；当 base 值为 0 时会根据情况选择用哪种进制：如果第一个字符是 0，就判断第二字符如果是 x 则用 16 进制，否则用 8 进制，第一个字符不是 0，则用 10 进制。一开始 strtoul() 会扫描参数 nptr 字符串，跳过前面的空格字符串，直到遇上数字或正负符号才开始做转换，再遇到非数字或字符串结束时 (') 结束转换，并将结果返回。若参数 endptr 不为 NULL，则会将遇到不合条件而终止的 nptr 中的字符指针由 endptr 返回。

strtol 使用如例 3.7 所示。

【例 3.7】 strtol 的使用举例。

```
#include<iostream>
#include<stdlib.h>
#include<string>
using namespace std;
int main(){
    char *endptr;
    char nptr[]="123abc";
    int ret = strtol(nptr, &endptr, 10 );
    cout<<"ret:"<<ret<<endl;
    cout<<"endptr:"<<endptr<<endl;
```

```

char *endptr2;
char nptr2[]=" \n\t abc";
ret = strtol(nptr2, &endptr2, 10 );
cout<<"ret:"<<ret<<endl;
cout<<"endptr2:"<<endptr2<<endl;

char *endptr8;
char nptr8[]="0123";
ret = strtol(nptr8, &endptr8, 0);
cout<<"ret:"<<ret<<endl;
cout<<"endptr8:"<<endptr8<<endl;

char *endptr16;
char nptr16[]="0x123";
ret = strtol(nptr16, &endptr16, 0);
cout<<"ret:"<<ret<<endl;
cout<<"endptr16:"<<endptr16<<endl;

return 0;
}

```

程序的执行结果是：

```

ret:123
endptr:abc
ret:0
endptr2:
    abc
ret:83
endptr8:
ret:291
endptr16:

```

例 3.7 中主要是使用 `strtol` 函数对不同的字符串取出整数。

```

char nptr[]="123abc";
int ret = strtol(nptr, &endptr, 10 );

```

十进制里没有“数字”a，所以扫描到a就结束，因此ret值是123，即endptr是abc。

```

char *endptr2;
char nptr2[]=" \n\t abc";
ret = strtol(nptr2, &endptr2, 10 );

```

由于函数会忽略nptr前面的空格，所以从字符a开始扫描，但是遇见的“第一个”即是不合法字符，所以函数结束，此时ret=0; endptr = nptr;

```

char *endptr8;
char nptr8[]="0123";
ret = strtol(nptr8, &endptr8, 0);
char *endptr16;

```

```
char nptr16[]="0x123";
ret = strtol(nptr16, &endptr16,0);
```

当第三个参数为 0 时，则分以下 3 种情况。

1) 如果 nptr 以 0x 开头，则把 nptr 当成 16 进制处理。

2) 如果 nptr 以 0 开头，则把 nptr 当成 8 进制处理。

3) 否则，把 nptr 当成 10 进制。

nptr8 是以 0 开头的，所以将 nptr8 当成 8 进制处理，再将 8 进制的 0123 转换为 10 进制的，得到了 $1*8^2+2*8+3=83$ 。nptr16 是以 0x 开头的，将 nptr16 当成 16 进制处理，再将 16 进制的 0x123 转换为 10 进制的，得到了 $1*16^2+2*16+3=291$ 。

因此，将字符串转换为 10 进制的数字，可以这么写：

```
static inline int64_t strtoint(const std::string& s){
    return strtoll(s.c_str(), 0, 10);
}
```

5. string 的其他常用成员函数

string 的其他常用成员函数有以下几个：

int capacity()const;	// 返回当前容量（即 string 中不必增加内存即可存放的元素个数）
int max_size()const;	// 返回 string 对象中可存放的最大字符串的长度
int size()const;	// 返回当前字符串的大小
int length()const;	// 返回当前字符串的长度
bool empty()const;	// 当前字符串是否为空
void resize(int len,char c);	// 把字符串当前大小置为 len，并用字符 c 填充不足的部分

3.3 vector

3.3.1 vector 是什么

vector 是线性容器，它的元素严格按照线性序列排序，和动态数组很相似。和数组类似的是，它的元素存储在一块连续的存储空间中，这也意味着不仅可以使使用迭代器（iterator）访问元素，还可以使用指针的偏移方式访问。和常规数组不一样的是，vector 能够自动存储元素，可以自动增长或缩小存储空间。

vector 的优点如下所述。

- (1) 可以使用下标访问个别的元素。
- (2) 迭代器可以按照不同的方式遍历容器。
- (3) 可以在容器的末尾增加或删除元素。

和数组相比，虽然容器在自动处理容量的大小时会消耗更多的内存，但是容器能提供和数组一样的性能，而且能很好地调整存储空间大小。

和其他标准的顺序容器相比，vector 能更有效访问容器内的元素和在末尾添加和删除

元素；而在其他位置添加和删除元素，vector 则不及其他顺序容器，在迭代器和引用也不比 lists 支持的好。

容器的大小和容器的容量是有区别的，大小是指元素的个数，容量是分配的内存大小，容量一般不小于容器的大小。vector::size() 返回容器的大小，vector::capacity() 返回容量值，容量多于容器大小的部分用于以防容器大小的增加使用。每次重新分配内存都会很影响程序的性能，所以一般分配的容量都大于容器的大小，若要自己指定分配的容量的大小，则可以使用 vector::reserve()，但是规定的值要大于 size() 值。

使用 vector 时需要包含的头文件 #include<vector>。

3.3.2 vector 的查增删

1. vector 的初始化和遍历

vector 的初始化方法如表 3-1 所示。

表 3-1 vector 的各种初始化方法

vector<T> v1	v1 是一个空 vector，它潜在的元素是 T 类型的，执行默认初始化
vector<T> v2(v1)	v2 中包含有 v1 所有元素的副本
vector<T> v2 = v1	等价于 v2(v1)，v2 中包含有 v1 所有元素的副本
vector<T> v3(n, val)	v3 包含了 n 个重复的元素，每个元素的值都是 val
vector<T> v4(n)	v4 包含了 n 个重复地执行了值初始化的对象
vector<T> v5{a,b,c...}	v5 包含了初始值个数的元素，每个元素被赋予相应的初始值
vector<T> v5={a,b,c...}	等价于 v5{a,b,c...}

vector 的遍历有 for(int i=0;i<a.size();++i)、for (iter=ivector.begin();iter!=ivector.end();iter++)、for_each 这几种方式。

【例 3.8】vector 的初始化和遍历。

```
#include <vector>
#include <iostream>
using namespace std;
int main(){
    int a[7]={1,2,3,4,5,6,7};
    vector<int> ivector(a,a+7);
    /*vector 的赋值并不可以像数组一样用花括号方便地完成赋值，这里借用了数组来初始化这个 vector
    初始化方式 vector<elementType> intvec(begin,end); 这样可以用起来看上去还是比较习惯的。*/
    vector<int>::iterator iter;
    for (iter=ivector.begin();iter!=ivector.end();iter++){
        cout<<*iter<<" ";
    }
    cout<<endl;
    ivector[5]=1;
    /*单个 vector 的赋值，这个方式看上去还是和数组一样的也可以这么写 ivector.at(5)=1; 但是
    就是不习惯 */
    cout<<ivector[5]<<endl<<ivector.size()<<endl;
```

```

    for (iter=ivector.begin();iter!=ivector.end();iter++){
        cout<<*iter<<" ";
    }
    cout<<endl;
    for(int i=0;i<5;i++){
        cout<<ivector[i]<<" ";
    }
    cout<<endl;
    return 0;
}

```

程序的执行结果是:

```

1 2 3 4 5 6 7
1
7
1 2 3 4 5 1 7
1 2 3 4 5

```

例 3.8 中展示了 `for(int i=0;i<a.size();++i)`、`for (iter=ivector.begin();iter!=ivector.end();iter++)` 的遍历方式, `vector` 中也可以直接用 `ivector[i]` 的方式访问第 `i` 个元素。

【例 3.9】 `for_each` 的遍历举例。

```

#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

void print(int n)
{
    cout<<n<<" ";
}

int main(){
    int a[7]={1,2,3,4,5,6,7};
    vector<int> ivector(a,a+7);
    vector<int>::iterator iter;
    for_each(ivector.begin(),ivector.end(),print);//用 for_each 进行遍历
    cout<<endl;
    ivector[5]=1;
    cout<<ivector[5]<<endl<<ivector.size()<<endl;
    for_each(ivector.begin(),ivector.end(),print);//用 for_each 进行遍历
    return 0;
}

```

程序的执行结果为:

```

1 2 3 4 5 6 7
1
7
1 2 3 4 5 1 7

```

例 3.9 中展示了如何使用 `for_each` 遍历 `vector` 中的元素。

`vector` 是个模板类，可以存放任何类型的对象。在 `vector` 中存放结构体时，可以按照自定义的排序方式排序。

【例 3.10】 `vector` 中存放结构体时的排序。

```
#include<algorithm>
#include<vector>
#include<iostream>
using namespace std;

typedef struct rect{
    int id;
    int length;
    int width;
    bool operator< (const rect &a) const{
        if(id!=a.id)
            return id<a.id;
        else{
            if(length!=a.length)
                return length<a.length;
            else
                return width<a.width;
        }
    }
}Rect;

int main(){
    vector<Rect> vec;
    Rect rect;
    rect.id=2;
    rect.length=3;
    rect.width=4;
    vec.push_back(rect);
    rect.id=1;
    rect.length=2;
    rect.width=3;
    vec.push_back(rect);
    vector<Rect>::iterator it=vec.begin();
    cout<<(*it).id<<' '<<(*it).length<<' '<<(*it).width<<endl;
    sort(vec.begin(),vec.end());
    cout<<(*it).id<<' '<<(*it).length<<' '<<(*it).width<<endl;
    return 0;
}
```

程序的执行结果是：

```
2 3 4
1 2 3
```

例 3.10 中，`vec` 中存放的是结构体 `Rect`。`vec` 未进行排序前，将会按照 `push_back` 的时

间顺序排序，并不会自动排序。排序后，可以按照结构体中对 rect 的重载方式进行排序：按照 id、length、width 升序排序，然后用 `<algorithm>` 中的 `sort` 函数排序。

除了重载结构体里的 `rect`，也可以在结构体外定义一个函数来进行比较。

【例 3.11】 结构体外定义比较函数。

```
#include<algorithm>
#include<vector>
#include<iostream>
using namespace std;

typedef struct rect{
    int id;
    int length;
    int width;
}Rect;

int cmp(Rect a,Rect b){
    if(a.id!=b.id)
        return a.id<b.id;
    else{
        if(a.length!=b.length)
            return a.length<b.length;
        else
            return a.width<b.width;
    }
}

int main(){
    vector<Rect> vec;
    Rect rect;
    rect.id=2;
    rect.length=3;
    rect.width=4;
    vec.push_back(rect);
    rect.id=1;
    rect.length=2;
    rect.width=3;
    vec.push_back(rect);
    vector<Rect>::iterator it=vec.begin();
    cout<<(*it).id<<' '<<(*it).length<<' '<<(*it).width<<endl;
    sort(vec.begin(),vec.end(),cmp);
    cout<<(*it).id<<' '<<(*it).length<<' '<<(*it).width<<endl;
    return 0;
}
```

程序的执行结果是：

```
2 3 4
1 2 3
```

例 3.11 与例 3.10 不同的地方是，例 3.11 中并没有对结构体进行重载，而是在结构体外

定义了一个比较函数；另外 `sort` 的调用方式也不相同，例 3.11 中加了个比较函数作为第 3 个参数。二者均可以实现对 `vec` 的排序。

2. vector 的查找

在 `vector` 中查找一个元素可以如例 3.12 所示。

【例 3.12】 在 `vector` 中查找元素。

```
#include<algorithm>
#include<vector>
#include<iostream>
using namespace std;
int main(){
    vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);
    vec.push_back(5);
    vector<int>::iterator iter=find(vec.begin(),vec.end(),3);
    if ( iter==vec.end())
        cout << "Not found" << endl;
    else
        cout << "Found" << endl;
    return 0;
}
```

程序的执行结果是：

```
Found
```

例 3.12 中，使用了 `find` 函数在 `vector` 中进行查找。注意 `find` 函数不属于 `vector` 的成员，而存在于算法中，所以应加上头文件 `#include <algorithm>`。

3. vector 的删除

`vector` 中的删除，可以有 `erase` 或 `pop_back` 函数。`erase` 可以删除指定元素或指定位置的元素，而 `pop_back` 只能去掉数组的最后一个数据。

`erase` 的函数原型有以下两种形式：

```
iterator erase(iterator position)。
iterator erase(iterator first, iterator last)。
```

假设有这样的程序：

```
vector<int> vec;
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);
```

```

vec.push_back(5);
for(vector<int>::iterator iter=veci.begin(); iter!=veci.end(); iter++){
    if( *iter == 3)
        veci.erase(iter);
}

```

乍一看这段代码很正常，其实这里面隐藏着一个很严重的错误：当 `veci.erase(iter)` 语句执行了之后，`iter` 就变成了一个野指针，对一个野指针进行 `iter++` 操作是肯定会出错的。

查看 MSDN，对于 `erase` 的返回值是这样描述的：An iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the vector if no such element exists，于是改代码：

```

for(vector<int>::iterator iter=vec.begin(); iter!=vec.end(); iter++){
    if( *iter == 3)
        iter = vec.erase(iter);
}

```

这段代码也是错误的：①无法删除两个连续的 3；②当数字 3 位于 `vector` 最后位置的时候，也会出错（在 `vec.end()` 上执行 `++` 操作）。正确的代码应如例 3.13 所示。

【例 3.13】 使用 `erase` 删除 `vector` 中某个元素。

```

#include<algorithm>
#include<vector>
#include<iostream>
using namespace std;
int main(){
    vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);
    vec.push_back(5);
    vector<int>::iterator iter=vec.begin();
    for(; iter!=vec.end(); ){
        if(*iter==3){
            iter=vec.erase(iter);
        }else{
            ++iter;
        }
    }
    for(iter=vec.begin(); iter!=vec.end(); iter++){
        cout<<*iter<<" ";
    }
    return 0;
}

```

程序的执行结果是：

1 2 4 5

例 3.13 中，for 语句条件里面删除元素时，返回值指向已删除元素的下一个位置，不是删除元素时则直接进行 ++ 操作。

使用 `vec.erase(vec.begin()+i,vec.end()+j)` 语句则是删除区间 `[i,j-1]` 间的元素。

而 `pop_back` 只能去掉数组的最后一个数据。

【例 3.14】vector 的 `pop_back` 函数使用举例。

```
#include<algorithm>
#include<vector>
#include<iostream>
using namespace std;
int main(){
    vector<int> vec;
    for(int i=0;i<10;i++)
        vec.push_back(i);
    vector<int>::iterator iter=vec.begin();
    for(iter=vec.begin();iter!=vec.end();iter++){
        cout<<*iter<<" ";
    }
    cout<<endl;
    vec.pop_back();
    for(iter=vec.begin();iter!=vec.end();iter++){
        cout<<*iter<<" ";
    }
    cout<<endl;
    return 0;
}
```

程序的执行结果是：

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8
```

例 3.14 中定义了一个存放整型数的 vector，里面存放了 0 ~ 9。用 `pop_back` 函数，则把最晚进入 vector 的 9 删除。

4. vector 的增加

vector 中的增加，可以有 `insert` 和 `push_back`。`insert` 是插入元素到某个位置中，`push_back` 是在最后添加一个元素。

`insert` 的函数原型有以下 3 种形式：

```
iterator insert( iterator loc, const TYPE &val );
// 在指定位置 loc 前插入值为 val 的元素，返回指向这个元素的迭代器
void insert( iterator loc, size_type num, const TYPE &val );
// 在指定位置 loc 前插入 num 个值为 val 的元素
void insert( iterator loc, input_iterator start, input_iterator end );
// 在指定位置 loc 前插入区间 [start, end) 的所有元素
```

【例 3.15】 vector 的查增删用法举例。

```

#include<algorithm>
#include<vector>
#include<iostream>
using namespace std;
void print( vector<int>v ){
    vector<int>::iterator iter=v.begin();
    for(;iter!=v.end();iter++)
        cout<<*iter<<" ";
    cout<<endl;
}
int main(){
    vector<int> v;                                // 现在容器中有 0 个元素
    int values[] = {1,3,5,7};
    v.insert(v.end(), values+1, values+3); // 现在容器中有 2 个元素分别为 :3,5
    print(v);
    v.push_back(9);                                // 现在容器中有 3 个元素分别为 :3,5,9
    print(v);
    v.erase(v.begin()+1);                          // 现在容器中有 2 个元素分别为 :3,9
    print(v);
    v.insert(v.begin()+1, 4);                      // 现在容器中有 3 个元素分别为 :3,4,9
    print(v);
    v.insert(v.end()-1, 4, 6);                    // 现在容器中有 7 个元素分别为 :3,4,6,6,6,6,9
    print(v);
    v.erase(v.begin()+1, v.begin()+3); // 现在容器中有 5 个元素分别为 :3,6,6,6,9
    print(v);
    v.pop_back();                                // 现在容器中有 4 个元素分别为 :3,6,6,6
    print(v);
    v.clear();                                    // 现在容器中有 0 个元素
    print(v);
    if (true == v.empty())                      // 如果容器为空则输出 "null"
    {
        std::cout<<"null"<<std::endl;
    }
    return 0;
}

```

例 3.15 的程序的执行结果如图 3-1 所示。

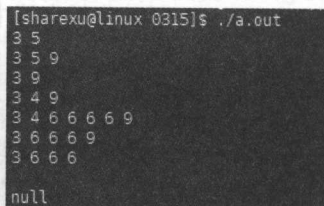
例 3.15 中的语句：

```
v.insert(v.end(), values+1, values+3);
```

就是将数组第 2 个元素和第 3 个元素的值插入到 v.end() 位置中，因为此时 v 还是空的，所以也就是往空 vector 里插入了两个元素。注意，这里只插入了两个元素，而没有插入 3 个元素。

```
v.erase(v.begin()+1);
```

v.begin() 是指第 1 个元素，那 v.begin()+1 就是指第 2 个元素，即这里是删除第 2 个元素。



```

[sharexu@linux 0315]$ ./a.out
3 5
3 5 9
3 9
3 4 9
3 4 6 6 6 6 9
3 6 6 6 9
3 6 6 6
null

```

图 3-1 例 3.15 程序的执行结果

```
v.insert(v.end()-1, 4, 6);
```

`v.end()` 是指最后一个元素的下一个位置, `v.end()-1` 就是倒数第 2 个元素的前面一个位置, 插入 4 个 6, 因此结果是 3 4 6 6 6 6 9。

程序中 `v.clear()` 表示将 `v` 清空; `v.empty()` 表示判断 `vector` 是否为空, 如果为空, 则返回 `true`。

3.3.3 vector 的内存管理与效率

1. 使用 `reserve()` 函数提前设定容量大小

关于 STL 容器, 最令人称赞的特性之一就是只要不超过它们的最大值, 就可以自动增长到足以容纳用户放进去的数据的大小。(这个最大容量值, 只要调用名叫 `max_size` 的成员函数就可以获得) 对于 `vector` 和 `string`, 如果需要更多空间, 就会以类似 `realloc` 的思想来增长大小。`vector` 容器支持随机访问, 因此为了提高效率, 它内部是使用动态数组的方式实现的。在通过 `reserve()` 函数来申请特定大小的内存空间时候总是按指数边界来增大其内部缓冲区。当进行 `insert` 或 `push_back` 等增加元素的操作时, 如果此时动态数组的内存不够用, 就要动态的重新分配当前大小的 1.5 ~ 2 倍的新内存区, 再把原数组的内容复制过去。所以, 在一般情况下, 其访问速度同一般数组相比, 只有在重新分配发生时, 其性能才会下降。正如例 3.15 中的代码一样, 进行 `pop_back` 操作时, `capacity` 并不会因为 `vector` 容器里的元素减少而有所下降, 还会维持操作之前的大小。对于 `vector` 容器来说, 如果有大量的数据需要进行 `push_back`, 应当使用 `reserve()` 函数提前设定其容量大小, 否则会出现许多次容量扩充操作, 导致效率低下。

`reserve` 成员函数允许开发者最小化必须进行的重新分配的次数, 因而可以避免真分配的开销和迭代器、指针、引用失效。但在解释 `reserve` 为什么可以那么做之前, 需要先简要介绍有时候令人困惑的 4 个相关成员函数, 如下所述。在标准容器中, 只有 `vector` 和 `string` 提供了所有这些函数。

(1) `size()` 可以获得容器有多少元素, 但不能获得容器为它容纳的元素分配的内存大小。

(2) `capacity()` 可以获得容器在它已经分配的内存中可以容纳多少元素。那是容器在那块内存中总共可以容纳多少元素, 而不是还可以容纳多少元素。如果想知道一个 `vector` 或 `string` 中有多少没有被占用的内存, 则必须从 `capacity()` 中减去 `size()`。如果 `size` 和 `capacity` 返回同样的值, 容器中就没有剩余空间了, 而下一次插入 (通过 `insert` 或 `push_back` 等) 会引发上面的重新分配步骤。

(3) `resize(Container::size_type n)` 用来强制把容器改为容纳 n 个元素。调用 `resize` 函数之后, `size` 函数将会返回 n 。如果 n 小于当前大小, 容器尾部的元素会被销毁。如果 n 大于当前大小, 新默认构造的元素会添加到容器尾部。如果 n 大于当前容量, 在元素加入之前会进行重新分配。

(4) `reserve(Container::size_type n)` 强制容器把它的容量改为不小于 n ，提供的 n 不小于当前所需的大小。因为容量需要增加，这一般会强迫进行一次重新分配。如果 n 小于当前容量，`vector` 会忽略它，则这个调用什么都不做，`string` 可能把它的容量减少为 `size()` 和 n 中大的数，但 `string` 的大小没有改变。

综上所述，只要有元素需要插入而且容器的容量不足时就会发生重新分配（包括它们维护的原始内存分配和回收，对象的拷贝和析构和迭代器、指针和引用的失效）。所以，避免重新分配的关键是使用 `reserve` 尽快把容器的容量设置为足够大，最好在容器被构造之后立刻进行。

例如，假定想建立一个容纳 1 ~ 1000 值的 `vector<int>`，若不使用 `reserve`，则可以像这样做：

```
vector<int> v;
for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

在大多数 STL 实现中，这段代码在循环过程中将会导致 2 ~ 10 次重新分配。（10 这个数没什么奇怪的。记住 `vector` 在重新分配发生时一般把容量翻倍，而 1000 约等于 2^{10} 。）

把代码改为使用 `reserve`，如下所示：

```
vector<int> v;
v.reserve(1000);
for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

则这在循环中不会发生重新分配。

由大小和容量之间的关系可以预言什么时候插入将引起 `vector` 或 `string` 执行重新分配，而且，可以预言什么时候插入会使指向容器中的迭代器、指针和引用失效。例如，给出这段代码：

```
string s;
...
if (s.size() < s.capacity()) {
    s.push_back('x');
}
```

`push_back` 的调用不会使指向这个 `string` 中的迭代器、指针或引用失效，因为 `string` 的容量保证大于它的大小；如果不是执行 `push_back`，代码在 `string` 的任意位置进行一个 `insert`，仍然可以保证在插入期间没有发生重新分配，但是，与伴随 `string` 插入时迭代器失效的一般规则一致，所有从插入位置到 `string` 结尾的迭代器、指针、引用将失效。

通常有两种情况使用 `reserve` 来避免不必要的重新分配。第一种可用的情况是当知道有多少元素将最后出现在容器中时，就像上面的 `vector` 代码，就可以提前 `reserve` 适当数量的空间；第二种情况是保留可能需要的最大的空间，然后添加完全部数据后，再修整掉任何多余的容量。

2. 使用“交换技巧”来修整 `vector` 过剩空间 / 内存

有一种方法来把它从曾经最大的容量减少到它现在需要的容量，这样的方法常常被称为

“收缩到合适”(shrink to fit)。该方法只需一条语句：`vector<int>(ivec).swap(ivec)`。

表达式 `vector<int>(ivec)` 表示建立一个临时 `vector`，它是 `ivec` 的一份拷贝。但是，`vector` 的拷贝构造函数只分配拷贝的元素需要的内存，所以这个临时 `vector` 没有多余的容量。然后临时 `vector` 和 `ivec` 交换数据完成，但 `ivec` 只有临时变量的修整过的容量，而这个临时变量则持有了曾经在 `ivec` 中的没用到的过剩容量。在这个语句结尾处，临时 `vector` 被销毁，以释放以前 `ivec` 使用的内存，收缩到合适的大小。

3. 用 swap 方法强行释放 vector 所占内存

```
template < class T> void ClearVector( vector<T>& v )
{
    vector<T>vtTemp;
    vtTemp.swap( v );
}

vector<int> v ;
nums.push_back(1);
nums.push_back(3);
nums.push_back(2);
nums.push_back(4);
vector<int>().swap(v);
/* 或者 v.swap(vector<int>()); */
/* 或者 { std::vector<int> tmp = v;    v.swap(tmp);    }; // 加大括号 { } 是让 tmp 退出 { }
// 时自动析构 */
```

4. Vector 内存管理成员函数的行为测试

【例 3.16】 `vector` 内存管理成员函数的行为测试。

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
    vector<int> iVec;
    cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

    /*1 个元素，容器容量为 1*/
    iVec.push_back(1);
    cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

    /*2 个元素，容器容量为 2*/
    iVec.push_back(2);
    cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

    /*3 个元素，容器容量为 4*/
    iVec.push_back(3);
    cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

    /*4 个元素，容器容量为 4*/
```



```

iVec.push_back(4);
cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

/*5 个元素, 容器容量为 8*/
iVec.push_back(5);
cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

/*6 个元素, 容器容量为 8*/
iVec.push_back(6);
cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

/*7 个元素, 容器容量为 8*/
iVec.push_back(7);
cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

/*8 个元素, 容器容量为 8*/
iVec.push_back(8);
cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

/*9 个元素, 容器容量为 16*/
iVec.push_back(9);
cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;

/* vs2005/8 容量增长不是翻倍的, 如
9 个元素 容量 9
10 个元素 容量 13 */

/* 测试 effective stl 中的特殊的交换 swap() */
cout<<" 容器大小:"<<iVec.size()<<" 容量:"<<iVec.capacity()<<endl;
vector<int>(iVec).swap(iVec);

cout<<" 临时的 vector<int> 对象的大小为:"<<(vector<int>(iVec)).size()<<endl;
cout<<" 临时的 vector<int> 对象的容量为:"<<(vector<int>(iVec)).capacity()<<endl;
cout<<" 交换后, 当前 vector 的大小为:"<<iVec.size()<<endl;
cout<<" 交换后, 当前 vector 的容量为:"<<iVec.capacity()<<endl;

return 0;
}

```

例 3.16 的程序执行结果如图 3-2 所示。

例 3.16 中展示了 vector 在不同情况下占用内存空间的大小情况: vector 是按照容器现在容量的一倍进行增长。vector 容器分配的是一块连续的内存空间, 每次容器的增长, 并不是在原有连续的内存空间后再进行简单的叠加, 而是重新申请一块更大的新内存, 并把现有容器中的元素逐个复制过去, 同时销毁旧的内存。这时原有指向旧内存空间的迭代器已经失效, 所以当操作容器时, 迭代器要及时更新。

```

容器大小:0容量:0
容器大小:1容量:1
容器大小:2容量:2
容器大小:3容量:4
容器大小:4容量:4
容器大小:5容量:8
容器大小:6容量:8
容器大小:7容量:8
容器大小:8容量:8
容器大小:9容量:16
容器大小:9容量:16
临时的vector<int>对象的大小为:9
临时的vector<int>对象的容量为:9
交换后,当前vector的大小为:9
交换后,当前vector的容量为:9

```

图 3-2 例 3.16 的程序执行结果

3.3.4 Vector 类的简单实现

实现一个 vector，绝对是 C++ 中的重点知识。下面例 3.13 中提供了类的简单实现。

【例 3.17】vector 类的简单实现。

```
#include<algorithm>
#include<iostream>
#include <assert.h>
using namespace std;
template<typename T>
class myVector
{
private:
    /*walk length*/
    /*myVector each time increase space length*/
    #define WALK_LENGTH 64;

public:
    /*default constructor*/
    myVector():array(0),theSize(0),theCapacity(0){}
    myVector(const T& t,unsigned int n):array(0),theSize(0),theCapacity(0){
        while(n--){
            push_back(t);
        }
    }

    /*copy constructor*/
    myVector(const myVector<T>& other):array(0),theSize(0),theCapacity(0){
        *this = other;
    }

    /*= operator*/
    myVector<T>& operator =(myVector<T>& other){
        if(this == &other)
            return *this;
        clear();
        theSize = other.size();
        theCapacity = other.capacity();
        array = new T[theCapacity];
        for(unsigned int i = 0 ;i<theSize;++i){
            array[i] = other[i];
        }
        return *this;
    }

    /*destructor*/
    ~myVector(){
        clear();
    }
}
```

```

/*the pos must be less than myVector.size();*/
T& operator[](unsigned int pos){
    assert(pos<theSize);
    return array[pos];
}

/*element theSize*/
unsigned int size(){
    return theSize;
}

/*alloc theSize*/
unsigned int capacity(){
    return theCapacity;
}

/*is empty*/
bool empty(){
    return theSize == 0;
}

/*clear myVector*/
void clear(){
    deallocator(array);
    array = 0;
    theSize = 0;
    theCapacity = 0;
}

/*adds an element in the back of myVector*/
void push_back(const T& t){
    insert_after(theSize-1,t);
}

/*adds an element int the front of myVector*/
void push_front(const T& t){
    insert_before(0,t);
}

/*inserts an element after the pos*/
/*the pos must be in [0,theSize);*/
void insert_after(int pos,const T& t){
    insert_before(pos+1,t);
}

/*inserts an element before the pos*/
/*the pos must be less than the myVector.size()*/
void insert_before(int pos,const T& t){
    if(theSize==theCapacity){
        T* oldArray = array;
        theCapacity += WALK_LENGTH;
    }
}

```

```

3.3.4 Vector array = allocator(theCapacity);
/*memcpy(array,oldArray,theSize*sizeof(T));*/
for(unsigned int i = 0 ;i<theSize;++i){
    array[i] = oldArray[i];
}
deallocater(oldArray);
}

for(int i = (int)theSize++;i>pos;--i){
    array[i] = array[i-1];
}
array[pos] = t;
}

/*erases an element in the pos;*/
/*pos must be in [0,theSize);*/
void erase(unsigned int pos){
    if(pos<theSize){
        --theSize;
        for(unsigned int i = pos;i<theSize;++i){
            array[i] = array[i+1];
        }
    }
}

private:
    T* allocator(unsigned int size){
        return new T[size];
    }

    void deallocater(T* arr){
        if(arr)
            delete[] arr;
    }

private:
    T* array;
    unsigned int theSize;
    unsigned int theCapacity;
};

void printfVector(myVector<int>& vector1){
    for(unsigned int i = 0 ; i < vector1.size();++i){
        cout<<vector1[i]<<" ";
    }
    cout<<"alloc size = "<<vector1.capacity()<<" ,size = "<<vector1.size()<<endl;
}

int main(){
    myVector<int> myVector1;
    myVector<int> myVector2(0,10);
    myVector2.push_front(1);

```

```

myVector2.erase(11);
printfVector(myVector2);
myVector1.push_back(2);
myVector1.push_front(1);
printfVector(myVector1);
myVector1.insert_after(1,3);
printfVector(myVector1);

myVector2 = myVector1;
myVector2.insert_before(0,0);
myVector2.insert_before(1,-1);
printfVector(myVector2);
return 0;
}

```

程序的执行结果为：

```

1,0,0,0,0,0,0,0,0,0,0,0,alloc size = 64,size = 11
1,2,alloc size = 64,size = 2
1,2,3,alloc size = 64,size = 3
0,-1,1,2,3,alloc size = 64,size = 5

```

STL 库中 `vector` 是一个自动管理的动态数组，只要明白 `vector` 的类型是一个数组，至于怎么去实现它其实不难。例 3.17 中选择了一种简单的方式去实现它：定义一个步长 `WALK_LENGTH`，在数组空间不够时，再重新申请长度为 `theCapacity + WALK_LENGTH` 的内存，这样就避免了每次当 `myVector` 元素增加的时候，需要去重新申请空间的问题，当然不好的地方就是会浪费一定的空间，但是时间效率上会提高很多。因为 `vector` 可以支持下标访问，所以就不用单独构造一个 `iterator`，从而提高效率。

例 3.17 中，`myVector` 拥有 3 个成员变量：元素的个数 `theSize`、容量 `theCapacity` 和一个指针数组 `array`。

默认构造函数里，把元素的个数 `theSize`、容量 `theCapacity` 都赋值为 0，数组赋值为空，代码如下：

```
myVector():array(0),theSize(0),theCapacity(0){ }
```

用几个相同的值赋值给 `myVector`，那应该是逐个添加的：

```

myVector(const T& t,unsigned int n):array(0),theSize(0),theCapacity(0){
    while(n--){
        push_back(t);
    }
}

```

进行重载：

```

myVector<T>& operator =(myVector<T>& other){
    if(this == &other)
        return *this;
}

```

```

clear();
theSize = other.size();
theCapacity = other.capacity();
array = new T[theCapacity];
for(unsigned int i = 0 ;i<theSize;++i)
{
    array[i] = other[i];
}
return *this;
}

```

如果参数与本 myVector 相同，那就无需赋值了；不相同时才需要赋值，并需要分别对 3 个成本变量进行赋值，元素个数、容量大小和数组内容。

析构函数里直接调用 clear 函数，如下所示：

```

~myVector(){
    clear();
}

```

用下标的方式访问 myVector 中的元素，其实就是访问数组 array 中的元素，注意下标必须小于元素个数，如下所示：

```

T& operator[](unsigned int pos){
    assert(pos<theSize);
    return array[pos];
}

```

获得元素个数和容器大小，直接返回成员变量即可，如下所示：

```

/*element theSize*/
unsigned int size(){
    return theSize;
}

/*alloc theSize*/
unsigned int capacity(){
    return theCapacity;
}

```

判断 myVector 是否为空，直接判断元素个数是否等于 0 即可，如下所示：

```

bool empty(){
    return theSize == 0;
}

```

清空 myVector 中的元素，需要删除掉数组指针，并把元素个数和容量大小都置 0，如下所示：

```

void clear(){
    deallocator(array);
}

```

```

    array = 0;
    theSize = 0;
    theCapacity = 0;
}

```

push_back、push_front 都可以归根于 insert，在哪个位置插入，如下所示：

```

void push_back(const T& t){
    insert_after(theSize-1,t);
}

```

```

void push_front(const T& t){
    insert_before(0,t);
}

```

```

void insert_after(int pos,const T& t){
    insert_before(pos+1,t);
}

```

```

void insert_before(int pos,const T& t){
    if(theSize==theCapacity){
        T* oldArray = array;
        theCapacity += WALK_LENGTH;
        array = allocator(theCapacity);
        /*memcpy(array,oldArray,theSize*sizeof(T));*/
        for(unsigned int i = 0 ;i<theSize;++i){
            array[i] = oldArray[i];
        }
        deallocator(oldArray);
    }
}

```

```

for(int i = (int)theSize++;i>pos;--i){
    array[i] = array[i-1];
}
array[pos] = t;
}

```

myVector 通过一个连续的数组存放元素，如果集合已满，在新增数据的时候，就要分配一块更大的内存，将原来的数据复制过来，释放之前的内存，再插入新增的元素。这个元素后面的所有元素都向后移动一个位置，在空出来的位置上存入新增的元素。

删除某个元素，则要把这个元素后面的都往前挪，并把元素个数 -1，如下所示：

```

void erase(unsigned int pos){
    if(pos<theSize){
        --theSize;
        for(unsigned int i = pos;i<theSize;++i){
            array[i] = array[i+1];
        }
    }
}

```


通过分析代码，可以发现 `vector` 的特点，如下所述。

- (1) 随即访问元素效率很高。
- (2) `push_back` 的效率也会很高。
- (3) `push_front` 的效率非常低，不建议使用。
- (4) `insert` 需要把插入位置以后的元素全部后移，效率比较低，不建议使用。
- (5) `erase` 需要把删除位置后面的元素全部前移，效率比较低，不建议使用。
- (6) 当内存不够时，需要重新申请内存，再把以前的元素复制过来，效率也比较低。

3.4 map

3.4.1 map 是什么

1. map 的本质

`map` 本质是一类关联式容器，属于模板类关联的本质在于元素的值与某个特定的键相关联，而并非通过元素在数组中的位置类获取。它的特点是增加和删除节点对迭代器的影响很小，除了操作节点，对其他的节点都没有什么影响。对于迭代器来说，不可以修改键值，只能修改其对应的实值。`map` 内部数据的组织，`map` 内部自建一棵红黑树（一种非严格意义上的平衡二叉树），这棵树具有对数据自动排序的功能，所以在 `map` 内部所有的数据都是有序的。

2. map 的功能

自动建立 Key-value 的一一对应关系。比如一个班级中，每个学生的学号跟他的姓名就存在着——映射的关系，这个模型用 `map` 可能轻易描述，很明显学号用 `int` 描述，姓名用字符串描述（本篇文章中不用 `char *` 来描述字符串，而是采用 STL 中 `string` 来描述），可以使用这样的一个 `map`：`Map<int, string> mapStudent;`

`key` 和 `value` 可以是任意你需要的类型，但是需要注意的是对于 `key` 的类型，唯一的约束就是必须支持 `<` 操作符。

根据 `key` 值快速查找记录，查找的复杂度基本是 $\log(N)$ ，即如果有 1000 个记录，最多查找 10 次；1 000 000 个记录，最多查找 20 次。除此之外，还有快速插入 Key-Value 记录、快速删除记录、根据 Key 修改 value 记录、遍历所有记录等功能。

3. map 需要包括的头文件

使用 `map` 得包含 `map` 类所在的头文件：`#include <map>` // 注意，STL 头文件没有扩展名 `.h`。

3.4.2 map 的查增删

1. map 的插入

先讲下 `map` 的插入，`map` 的插入有 3 种方式：用 `insert` 函数插入 pair 数据、用 `insert` 函数插入 `value_type` 数据和用数组方式插入数据。

【例 3.18】 用 insert 函数插入 pair 数据。

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main()
{
    map<int, string> mapStudent;
    mapStudent.insert(pair<int, string>(1, "student_one"));
    mapStudent.insert(pair<int, string>(2, "student_two"));
    mapStudent.insert(pair<int, string>(3, "student_three"));
    map<int, string>::iterator iter;
    for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++){
        cout<<iter->first<<" "<<iter->second<<endl;
    }
    return 0;
}
```

程序的执行结果是：

```
1 student_one
2 student_two
3 student_three
```

例 3.18 中，声明了一个 key 为 int 类型，value 为 string 类型的 map，用 insert 函数插入 pair 数据，且需要在 insert 的参数中将 (1, "student_one") 转换为 pair 数据再进行插入。

【例 3.19】 用 insert 函数插入 value_type 数据。

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main()
{
    map<int, string> mapStudent;
    mapStudent.insert(map<int, string>::value_type (1,"student_one"));
    mapStudent.insert(map<int, string>::value_type (2,"student_two"));
    mapStudent.insert(map<int, string>::value_type (3,"student_three"));
    map<int, string>::iterator iter;
    for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++){
        cout<<iter->first<<" "<<iter->second<<endl;
    }
    return 0;
}
```

程序的执行结果是：

```
1 student_one
2 student_two
3 student_three
```

例 3.19 中，声明了一个 key 为 int 类型，value 为 string 类型的 map，用 insert 函数插入 value_type 数据，且需要在 insert 的参数中将 (1, "student_one") 转换为 map<int, string>::value_type 数据再进行插入。

【例 3.20】map 中用数组方式插入数据。

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main(){
    map<int, string> mapStudent;
    mapStudent[1] = "student_one";
    mapStudent[2] = "student_two";
    mapStudent[3] = "student_three";
    map<int, string>::iterator iter;
    for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++){
        cout<<iter->first<<" "<<iter->second<<endl;
    }
    return 0;
}
```

程序的执行结果是：

```
1 student_one
2 student_two
3 student_three
```

例 3.20 中展示了用数组方式在 map 中插入数据，和数组访问一样，有下标、直接赋值。以上 3 种用法，虽然都可以实现数据的插入，但是它们是有区别的，当然了第一种和第二种在效果上是完成一样的，用 insert 函数插入数据，在数据的插入上涉及集合的唯一性这个概念，即当 map 中有这个关键字时，insert 操作是插入数据不了的，但是用数组方式就不同了，它可以覆盖以前该关键字对应的值。

```
mapStudent.insert(map<int, string>::value_type (1, "student_one"));
mapStudent.insert(map<int, string>::value_type (1, "student_two"));
```

上面这两条语句执行后，map 中 1 这个关键字对应的值是 student_one，第二条语句并没有生效，那么这就涉及如何知道 insert 语句是否插入成功的问题了，可以用 pair 来获得是否插入成功，程序如下：

```
pair<map<int, string>::iterator, bool> insert_pair;
insert_pair = mapStudent.insert(map<int, string>::value_type (1, "student_one"));
```

可以通过 pair 的第二个变量来知道是否插入成功，它的第一个变量返回的是一个 map 的迭代器，如果插入成功的话 insert_Pair.second 应该是 true 的，否则为 false。

【例 3.21】用 pair 判断 insert 到 map 的数据是否插入成功。

```

#include <map>
#include <string>
#include <iostream>
using namespace std;
int main(){
    map<int, string> mapStudent;
    pair<map<int, string>::iterator, bool> insert_pair;
    insert_pair = mapStudent.insert(pair<int, string>(1, "student_one"));
    if(insert_pair.second == true){
        cout<<"Insert Successfully"<<endl;
    }
    else{
        cout<<"Insert Failure"<<endl;
    }
    insert_pair = mapStudent.insert(pair<int, string>(1, "student_two"));
    if(insert_pair.second == true){
        cout<<"Insert Successfully"<<endl;
    }else{
        cout<<"Insert Failure"<<endl;
    }
    map<int, string>::iterator iter;
    for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++){
        cout<<iter->first<<" "<<iter->second<<endl;
    }
    return 0;
}

```

程序的执行结果是：

```

Insert Successfully
Insert Failure
1 student_one

```

例 3.21 中，用 pair 判断 insert 到 map 的数据是否插入成功。pair 变量 insert_pair 中的第一个元素的类型是 map<int, string>::iterator，是和即将要判断的 map 中的 key、value 类型一致的一个 map 的迭代器。如果 insert 成功了，则 insert_pair.second 的结果为 true，否则则为 false。同一个 key 已经有数据之后，再 insert 就会失败。而数组插入的方式，则是直接覆盖。

【例 3.22】 数据方式插入 map 覆盖原有的数据。

```

#include <map>
#include <string>
#include <iostream>
using namespace std;
int main()
{
    map<int, string> mapStudent;
    mapStudent[1] = "student_one";
    mapStudent[1] = "student_two";
    mapStudent[2] = "student_three";
}

```

```

map<int, string>::iterator iter;
for(iter = mapStudent.begin(); iter != mapStudent.end(); iter++){
    cout<<iter->first<<" "<<iter->second<<endl;
}
return 0;
}

```

程序的执行结果是：

```

1 student_two
2 student_three

```

例 3.22 中展示了 mapStudent[1] 上已经有数据 "student_one" 了，再用语句：

```
mapStudent[1] = "student_two";
```

就可以直接覆盖成功。

2. map 的遍历

map 数据的遍历，这里也提供 3 种方法，来对 map 进行遍历：应用前向迭代器方式、应用反向迭代器方式和数组方式。应用前向迭代器，上面举例程序中已经讲解过了，这里着重讲解应用反向迭代器的方式，下面举例说明。

【例 3.23】map 反向迭代器的使用举例。

```

#include <map>
#include <string>
#include <iostream>
using namespace std;
int main(){
    map<int, string> mapStudent;
    mapStudent[1] = "student_one";
    mapStudent[2] = "student_two";
    mapStudent[3] = "student_three";
    map<int, string>::reverse_iterator iter;
    for(iter = mapStudent.rbegin(); iter != mapStudent.rend(); iter++){
        cout<<iter->first<<" "<<iter->second<<endl;
    }
    return 0;
}

```

程序的执行结果是：

```

3 student_three
2 student_two
1 student_one

```

例 3.23 中，iter 就是一个反向迭代器 reverse_iterator，它需要使用 rbegin() 和 rend() 方法指出反向遍历的起始位置和终止位置。注意，前向遍历的一般是从 begin() 到 end() 遍历，而反向遍历则是从 rbegin() 到 rend()。

【例 3.24】 用数组方式遍历 map。

```

#include<map>
#include<string>
#include<iostream>
using namespace std;
int main(){
    map<int,string> mapStudent;
    mapStudent[1] = "student_one";
    mapStudent[2] = "student_two";
    mapStudent[3] = "student_three";
    int iSize = mapStudent.size();
    for(int i = 1; i <= iSize; i++){
        cout<<i<<" "<<mapStudent[i]<<endl;
    }
    return 0;
}

```

例 3.24 中，用 size() 方法确定当前 map 中有多少元素。用数组访问 vector 时，下标是从 0 ~ (size-1)，而用数组访问 map，却是从 1~size，这是有所不同的，请使用者多加注意。

3. map 的查找

在这里可以充分体会到 map 在数据插入时保证有序的好处。要判定一个数据（关键字）是否在 map 中出现的方法比较多，这里给出 2 种常用的数据查找方法。

第一种：用 count 函数来判定关键字是否出现，其缺点是无法定位数据出现位置，由于 map 的一对一的映射特性，就决定了 count 函数的返回值只有两个，要么是 0，要么是 1，当要判定的关键字出现时返回 1。

第二种：用 find 函数来定位数据出现位置，它返回的一个迭代器，当数据出现时，它返回数据所在位置的迭代器；如果 map 中没有要查找的数据，它返回的迭代器等于 end 函数返回的迭代器。

【例 3.25】 用 find 方法查找 map 中的数据。

```

#include<map>
#include<string>
#include<iostream>
using namespace std;
int main(){
    map<int,string> mapStudent;
    mapStudent[1] = "student_one";
    mapStudent[2] = "student_two";
    mapStudent[3] = "student_three";
    map<int, string>::iterator iter=mapStudent.find(1);
    if(iter != mapStudent.end()){
        cout<<"Found, the value is "<<iter->second<<endl;
    }else{
        cout<<"Do not found"<<endl;
    }
}

```



```

    }
    return 0;
}

```

程序的执行结果是：

```
Find, the value is student_one
```

程序的执行结果是：



注意 find 函数返回的是一个迭代器；找不到对应数据的时候，则会返回 mapStudent.end()。

4. map 的删除

用 erase 方法可删除 map 中的元素。erase 的函数原型是：

```
map.erase(k)
```

删除 map 中键为 k 的元素，并返回 size_type 类型的值以表示删除的元素个数，代码如下：

```
map.erase(p)
```

从 map 中删除迭代器 p 所指向的元素。p 必须指向 map 中确实存在的元素，而且不能等于 map.end()，返回 void 类型，代码如下：

```
map.erase(b,e)
```

从 map 中删除一段范围内的元素，该范围由迭代器对 b 和 e 标记。b 和 e 必须标记 map 中的一段有效范围：即 b 和 e 都必须指向 map 中的元素或最后一个元素的下一个位置。而且，b 和 e 要么相等（此时删除的范围为空），要么 b 所指向的元素必须出现在 e 所指向的元素之前，返回 void 类型。常用的是第二种，并且是在遍历的过程中删除元素。

【例 3.26】 用 erase 方法删除 map 中的元素。

```

#include <map>
#include <string>
#include <iostream>
using namespace std;
int main(){
    map<int, string> mapStudent;
    mapStudent[1]="student_one";
    mapStudent[2]="student_two";
    mapStudent[3]="student_three";
    mapStudent[4]="student_four";
    map<int, string>::iterator iter=mapStudent.begin();
    for(;iter!=mapStudent.end();){
        if((*iter).second=="student_one"){
            mapStudent.erase(iter++);
        }
        else{
            ++iter;
        }
    }
}

```

```

    }
    for(iter=mapStudent.begin();iter!=mapStudent.end();iter++){
        cout<<iter->first<<" "<<iter->second<<endl;
    }
    return 0;
}

```

程序的执行结果是:

```

2 student_two
3 student_three
4 student_four

```



注意 mapStudent.erase(iter++); 中的 iter++, 不是 erase(iter), 然后 iter++. 因为 iter 指针被 erase 之后就失效了, 不能再用 iter++; 也不是 erase(++iter), 这样就不是删 iter 原来指向的元素了。

5. map 的排序

map 的排序默认按照 key 从小到大排序, 但有以下几点需要注意: ①按照 key 从大到小排序; ②key (第一个元素) 是一个结构体; ③想按 value (第二个元素) 排序。

map 是 STL 里面的一个模板类, 现在来看下 map 的定义:

```

template < class Key, class T, class Compare = less<Key>,
          class Allocator = allocator<pair<const Key,T> > > class map;

```

它有 4 个参数, 其中比较熟悉的有两个: Key 和 Value。第 4 个是 Allocator, 用来定义存储分配模型的, 此处不作介绍。

现在重点看下第 3 个参数:

```

class Compare = less<Key>

```

这也是一个 class 类型的, 而且提供了默认值 less<Key>。less 是 STL 里面的一个函数对象, 那么什么是函数对象呢?

所谓的函数对象, 即调用操作符的类, 其对象常称为函数对象 (function object), 它们是行为类似函数的对象。表现出一个函数的特征, 就是通过“对象名 +(参数列表)”的方式使用一个类, 其实质是对 operator() 操作符的重载。

现在来看一下 less 的实现:

```

template <class T> struct less : binary_function <T,T,bool> {
    bool operator() (const T& x, const T& y) const
    {return x<y;}
};

```

它是一个带模板的 struct，里面仅仅对 () 运算符进行了重载，实现很简单，但用起来很方便，这就是函数对象的优点所在。STL 中还为四则运算等常见运算定义了这样的函数对象，与 less 相对的还有 greater：

```
template <class T> struct greater : binary_function<T,T,bool> {
    bool operator()(const T& x, const T& y) const
    {return x>y;}
};
```

因此，要想让 map 中的元素按照 key 从大到小排序，可以如例 3.27 所示。

【例 3.27】 让 map 中的元素按照 key 从大到小排序。

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main(){
    map<string, int, greater<string> > mapStudent;
    mapStudent["LiMin"]=90;
    mapStudent["ZiLinMi"]=72;
    mapStudent["BoB"]=79;
    map<string, int>::iterator iter=mapStudent.begin();
    for(iter=mapStudent.begin();iter!=mapStudent.end();iter++){
        cout<<iter->first<<" "<<iter->second<<endl;
    }
    return 0;
}
```

程序的执行结果是：

ZiLinMi 72

LiMin 90

BoB 79

如例 3.27 中所示，只需指定它的第 3 个参数 Compare，把默认的 less 指定为 greater，即可达到按照 key 从大到小排序。现在知道如何为 map 指定 Compare 类了，如果想自己写一个 Compare 的类，让 map 按照想要的顺序来存储，比如按照学生姓名的长短排序进行存储，那么只要自己写一个函数对象，实现想要的逻辑，并在定义 map 的时候把 Compare 指定为自己编写的这个就可以实现了，代码如下：

```
struct CmpByKeyLength {
    bool operator()(const string& k1, const string& k2) {
        return k1.length() < k2.length();
    }
};
```

这里不用把 Compare 定义为模板，直接指定它的参数为 string 类型就可以了。

【例 3.28】 重定义 map 内部的 Compare 函数。

```

#include <map>
#include <string>
#include <iostream>
using namespace std;
struct CmpByKeyLength {
    bool operator()(const string& k1, const string& k2) {
        return k1.length() < k2.length();
    }
};
int main(){
    map<string, int, CmpByKeyLength > mapStudent;
    mapStudent["LiMin"]=90;
    mapStudent["ZiLinMi"]=72;
    mapStudent["BoB"]=79;
    map<string, int>::iterator iter=mapStudent.begin();
    for(iter=mapStudent.begin();iter!=mapStudent.end();iter++){
        cout<<iter->first<<" "<<iter->second<<endl;
    }
    return 0;
}

```

程序的执行结果是：

```

BoB 79
LiMin 90
ZiLinMi 72

```

因此，想改变 map 的 key 排序方法，可以通过修改 Compare 函数实现。

key 是结构体的，如果没有重载小于号 (<) 操作，就会导致 insert 函数在编译时就无法编译成功。其实，为了实现快速查找，map 内部本身就是按序存储的（比如红黑树）。在插入 <key, value> 键值对时，就会按照 key 的大小顺序进行存储。这也是作为 key 的类型必须能够进行 < 运算比较的原因。

【例 3.29】 key 是结构体的 map 排序。

```

#include <map>
#include <string>
#include <iostream>
using namespace std;
typedef struct tagStudentInfo
{
    int iID;
    string strName;
    bool operator < (tagStudentInfo const& r) const {
        // 这个函数指定排序策略，按 iID 排序，如果 iID 相等的话，按 strName 排序
        if(iID < r.iID) return true;
        if(iID == r.iID) return strName.compare(r.strName) < 0;
        return false;
    }
}

```

```

    }
}StudentInfo;// 学生信息
int main(){
    /* 用学生信息映射分数 */
    map<StudentInfo, int>mapStudent;
    StudentInfo studentInfo;
    studentInfo.iID = 1;
    studentInfo.strName = "student_one";
    mapStudent[studentInfo]=90;
    studentInfo.iID = 2;
    studentInfo.strName = "student_two";
    mapStudent[studentInfo]=80;
    map<StudentInfo, int>::iterator iter=mapStudent.begin();
    for(;iter!=mapStudent.end();iter++){
        cout<<iter->first.iID<<" "<<iter->first.strName<<" "<<iter->second<<endl;
    }
    return 0;
}

```

程序的执行结果是：

```

1 student_one 90
2 student_two 80

```

例 3.29 中，mapStudent 的 key 是 StudentInfo 类型的，要重载 StudentInfo 类型的 < 号，这样才能正常地插入到 mapStudent 中。

如果想按照 map 的 value（第二个元素）排序，第一反应是利用 stl 中提供的 sort 算法实现，这个想法是好的，不幸的是，sort 算法有个限制，利用 sort 算法只能对序列容器进行排序，只能是线性的（如 vector、list、deque）。map 也是一个集合容器，它里面存储的元素是 pair，但是它不是线性存储的（如红黑树），所以利用 sort 不能直接和 map 结合进行排序。虽然不能直接用 sort 对 map 进行排序，那么可以间接进行，把 map 中的元素放到序列容器（如 vector）中，然后再对这些元素进行排序呢？这个想法看似是可行的。要对序列容器中的元素进行排序，也有个必要条件：就是容器中的元素必须是可比较的，也就是实现了 < 操作的。那么现在就来看下 map 中的元素是否满足这个条件。

已知 map 中的元素类型为 pair，具体定义如下：

```

template <class T1, class T2> struct pair
{
    typedef T1 first_type;
    typedef T2 second_type;

    T1 first;
    T2 second;
    pair() : first(T1()), second(T2()) {}
    pair(const T1& x, const T2& y) : first(x), second(y) {}
    template <class U, class V>
        pair (const pair<U,V> &p) : first(p.first), second(p.second) { }
}

```

pair 也是一个模板类，这样就实现了良好的通用性。它仅有两个数据成员 first 和 second，即 key 和 value，而且在 <utility> 头文件中，还为 pair 重载了 < 运算符，具体实现如下所示：

```
template<class _T1, class _T2>
inline bool
operator<(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
{ return __x.first < __y.first
    || (!(__y.first < __x.first) && __x.second < __y.second); }
```

重点看下其实现，如下所示：

```
__x.first < __y.first || (!(__y.first < __x.first) && __x.second < __y.second)
```

这个 less 在两种情况下返回 true。第一种情况：__x.first < __y.first，这种情况较好理解，就是比较 key 与 __x、__y 的大小，如果 __x 的 key 小于 __y 的 key 则返回 true。

第二种情况有点费解，代码如下所示：

```
!(__y.first < __x.first) && __x.second < __y.second
```

当然由于 || 运算具有短路作用，即当前面的条件不满足时，才进行第二种情况的判断。第一种情况 __x.first < __y.first 不成立，即 __x.first >= __y.first 成立，在这个条件下，再来分析下 !(__y.first < __x.first) && __x.second < __y.second。

!(__y.first < __x.first) 表示 y 的 key 不小于 x 的 key，结合前提条件，__x.first < __y.first 不成立，即 x 的 key 不小于 y 的 key。

即：!(__y.first < __x.first) && !(__x.first < __y.first) 等价于 __x.first == __y.first，也就是说，第二种情况是在 key 相等的情况下，比较两者的 value (second)。

这里比较令人费解的地方就是，为什么不直接写 __x.first == __y.first 呢？这么写看似费解，但其实也不无道理：前面讲过，作为 map 的 key 必须实现 < 操作符的重载，但是并不保证 == 操作符也被重载了，如果 key 没有提供 ==，那么 __x.first == __y.first 的写法就不对。由此可见，STL 中的代码是相当严谨的，值得好好研读。

现在知道了 pair 类重载了 < 符，但是它并不是按照 value 进行比较的，而是先对 key 进行比较，key 相等时候才对 value 进行比较。显然不能满足按 value 进行排序的要求。

而且，既然 pair 已经重载了 < 符，但不能修改其实现，也不能在外部重复实现重载 < 符。

如果 pair 类本身没有重载 < 符，那么按照下面的代码重载 < 符，是可以实现对 pair 类按 value 比较的。但现在这样做不行了，甚至会出错（编译器不同，严格的就报错）。

```
typedef pair<string, int> PAIR;
bool operator< (const PAIR& lhs, const PAIR& rhs) {
    return lhs.second < rhs.second;
}
```


那么要如何实现对 pair 按 value 进行比较呢？第一种是最原始的方法，写一个比较函数；第二种刚才用到了，写一个函数对象，如下所示。这两种方式实现起来都比较简单。

```
typedef pair<string, int> PAIR;
bool cmp_by_value(const PAIR& lhs, const PAIR& rhs) {
    return lhs.second < rhs.second;
}
struct CmpByValue {
    bool operator()(const PAIR& lhs, const PAIR& rhs) {
        return lhs.second < rhs.second;
    }
};
```

接下来使用 sort 算法，来检验其是不是像 map 一样，也可以对指定的元素进行比较，代码如下所示：

```
template <class RandomAccessIterator>
void sort ( RandomAccessIterator first, RandomAccessIterator last );
template <class RandomAccessIterator, class Compare>
void sort ( RandomAccessIterator first, RandomAccessIterator last, Compare comp );
```

结果表明，sort 算法和 map 一样，也可以对指定元素进行比较，即指定 Compare。需要注意的是，map 是在定义时指定的，所以传参的时候直接传入函数对象的类名，就像指定 key 和 value 时指定的类型名一样；sort 算法是在调用时指定的，需要传入一个对象，当然这个也简单，类名 () 就会调用构造函数生成对象。

这里也可以传入一个函数指针，就是把上面说的第一种方法的函数名传过来。

【例 3.30】将 map 按 value 排序。

```
#include <map>
#include <vector>
#include <string>
#include <iostream>
using namespace std;
typedef pair<string, int> PAIR;
bool cmp_by_value(const PAIR& lhs, const PAIR& rhs) {
    return lhs.second < rhs.second;
}
struct CmpByValue {
    bool operator()(const PAIR& lhs, const PAIR& rhs) {
        return lhs.second < rhs.second;
    }
};
int main(){
    map<string, int> name_score_map;
    name_score_map["LiMin"] = 90;
    name_score_map["ZiLinMi"] = 79;
    name_score_map["BoB"] = 92;
    name_score_map.insert(make_pair("Bing", 99));
```

```

name_score_map.insert(make_pair("Albert",86));
/* 把 map 中元素转存到 vector 中 */
vector<PAIR> name_score_vec(name_score_map.begin(), name_score_map.end());
sort(name_score_vec.begin(), name_score_vec.end(), CmpByValue());
/*sort(name_score_vec.begin(), name_score_vec.end(), cmp_by_value); 也是可以的 */
for (int i = 0; i != name_score_vec.size(); ++i) {
    cout<<name_score_vec[i].first<<" "<<name_score_vec[i].second<<endl;
}
return 0;
}

```

例 3.30 中要对 map 中的 value 进行排序, 先把 map 的元素按 pair 形式插入到 vector 中, 再对 vector 进行排序 (用一个新写的比较函数), 这样就可以实现按 map 的 value 排序了。

3.4.3 map 的原理

map 内部自建一棵红黑树 (一种非严格意义上的平衡二叉树), 这棵树具有对数据自动排序的功能, 所以在 map 内部所有的数据都是有序的。这里简单讲下红黑树的含义。

先来看下算法导论对 R-B Tree 的介绍: 红黑树, 一种二叉查找树, 但在每个结点上增加一个存储位表示结点的颜色, 可以是 Red 或 Black。通过对任何一条从根到叶子的路径上各个结点着色方式的限制, 红黑树确保没有一条路径会比其他路径长出两倍, 因而是接近平衡的。红黑树, 作为一棵二叉查找树, 满足二叉查找树的一般性质。下面, 了解下二叉查找树的一般性质。

二叉查找树, 也称有序二叉树 (ordered binary tree), 或已排序二叉树 (sorted binary tree), 是指一棵空树或者具有下列性质的二叉树: ①若任意节点的左子树不空, 则左子树上所有结点的值均小于它的根结点的值; ②若任意节点的右子树不空, 则右子树上所有结点的值均大于它的根结点的值; ③任意节点的左、右子树也分别为二叉查找树; ④没有键值相等的节点 (no duplicate node)。

因为一棵由 n 个结点随机构造的二叉查找树的高度为 $\lg n$, 所以二叉查找树的一般操作的执行时间为 $O(\lg n)$ 。但二叉查找树若退化成了一棵具有 n 个结点的线性链后, 则这些操作最坏情况运行时间为 $O(n)$ 。

红黑树虽然本质上是一棵二叉查找树, 但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡, 从而保证了红黑树的查找、插入、删除的时间复杂度最坏为 $O(\lg n)$ 。

它是如何保证一棵 n 个结点的红黑树的高度始终保持在 $\lg n$ 的呢? 这就引出了红黑树的 5 个性质: ①每个结点要么是红的要么是黑的; ②根结点是黑的; ③每个叶结点 (叶结点即指树尾端 NIL 指针或 NULL 结点) 都是黑的; ④如果一个结点是红的, 那么它的两个儿子都是黑的; ⑤对于任意结点而言, 其到叶结点树尾端 NIL 指针的每条路径都包含相同数目的黑结点。

正是红黑树的这 5 条性质，使一棵 n 个结点的红黑树始终保持了 $\log n$ 的高度，从而也就解释了上面所说的“红黑树的查找、插入、删除的时间复杂度最坏为 $O(\log n)$ ”这一结论成立的原因。

如图 3-3 所示，是一颗红黑树（图 3-3 引自 wikipedia: <http://t.cn/hgvH11>）。

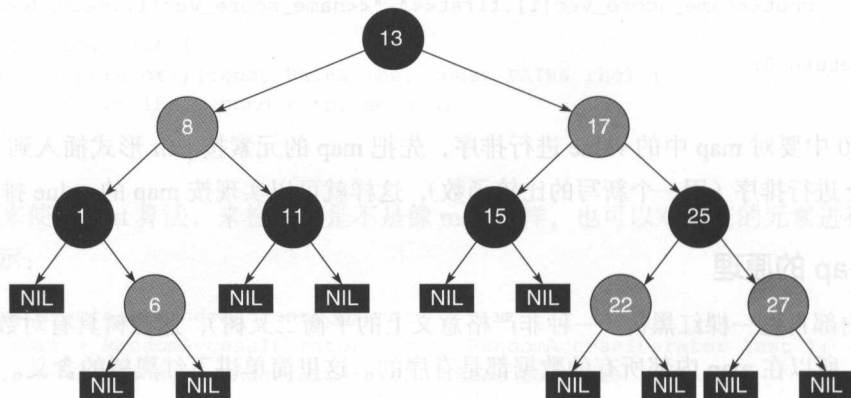


图 3-3 红黑树

当在对红黑树进行插入和删除等操作时，对树做了修改可能会破坏红黑树的性质。为了继续保持红黑树的性质，可以通过对结点进行重新着色，以及对树进行相关的旋转操作，即通过修改树中某些结点的颜色及指针结构，来达到对红黑树进行插入或删除结点等操作后继续保持它的性质或平衡的目的。

树的旋转分为左旋和右旋，下面借助图来介绍一下左旋和右旋这两种操作。

1) 左旋操作（图 3-4）。

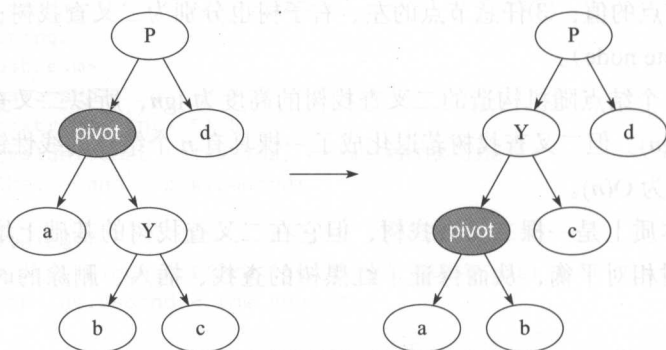


图 3.4 左旋操作示意图

如图 3-4 所示，当在某个结点 pivot 上，做左旋操作时，假设它的右孩子 y 不是 NIL[T]，pivot 可以为任何不是 NIL[T] 的左子结点。左旋以 pivot 到 Y 之间的链为“支轴”进行，它使 Y 成为该子树的新根，而 Y 的左孩子 b 则成为 pivot 的右孩子，在代码中表示如下所示。

```

LeftRotate(T, x)
y ← x.right           // y 是 x 的右孩子
x.right ← y.left       // y 的左孩子成为 x 的右孩子
if y.left ≠ T.nil
    y.left.p ← x
y.p ← x.p              // y 成为 x 的父亲
if x.p = T.nil
    then T.root ← y
else if x = x.p.left
    then x.p.left ← y
else x.p.right ← y
y.left ← x              // x 作为 y 的左孩子
x.p ← y

```

2) 右旋操作 (图 3-5)。

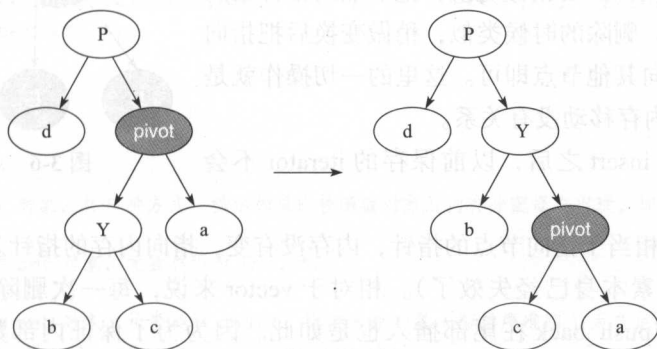


图 3-5 右旋操作示意图

右旋与左旋差不多，再此不做详细介绍。

树在经过左旋右旋之后，树的搜索性质保持不变，但树的红黑性质被破坏了，所以红黑树插入和删除数据后，需要利用旋转与颜色重涂来重新恢复树的红黑性质。

由于 map 的内部实现过于复杂，本章不做详细介绍，有兴趣的读者可自行在网上查资料。

3.5 set

3.5.1 set 是什么

C++ STL 之所以得到广泛的赞誉，也被很多人使用，不只是提供了像 vector、string、list 等方便的容器，更重要的是 STL 封装了许多复杂的数据结构算法和大量常用数据结构操作。vector 封装数组，list 封装了链表，map 和 set 封装了二叉树等，在封装这些数据结构的时候，STL 按照程序员的使用习惯，以成员函数方式提供的常用操作，如：插入、排序、删除、查找等。让用户在 STL 使用过程中，并不会感到陌生。

关于 set，必须说明的是 set 关联式容器。set 作为一个容器也是用来存储同一数据类型

的数据类型，并且能从一个数据集合中取出数据，在 `set` 中每个元素的值都唯一的，而且系统能根据元素的值自动进行排序。应该注意的是 `set` 中数元素的值不能直接被改变。C++ STL 中标准关联容器 `set`、`multiset`、`map`、`multimap` 内部采用的都是红黑树。红黑树的统计性能要好于一般平衡二叉树，所以被 STL 选择作为关联容器的内部结构。

关于 `set` 有下面几个问题需要注意。

(1) 为何 `map` 和 `set` 的插入删除效率比用其他序列容器高？

表面上看，因为对于关联容器来说，不需要做内存拷贝和内存移动。`set` 容器内所有元素都是以节点的方式来存储，其节点结构和链表差不多，指向父节点和子节点，`set` 的结构图如图 3-6 所示。

因此插入的时候只需要稍做变换，把节点的指针指向新的节点就可以了。删除的时候类似，稍做变换后把指向删除节点的指针指向其他节点即可。这里的一切操作就是指针换来换去，和内存移动没有关系。

(2) 为何每次 `insert` 之后，以前保存的 `iterator` 不会失效？

`iterator` 这里就相当于指向节点的指针，内存没有变，指向内存的指针怎么会失效呢（当然被删除的那个元素本身已经失效了）。相对于 `vector` 来说，每一次删除和插入，指针都有可能失效，调用 `push_back` 在尾部插入也是如此。因为为了保证内部数据的连续存放，`iterator` 指向的那块内存存在删除和插入过程中可能已经被其他内存覆盖或者内存已经被释放了。即使用 `push_back` 的时候，容器内部空间可能不够，需要一块新的更大的内存，只有把以前的内存释放，并申请新的更大的内存，并复制已有的数据元素到新的内存，最后把需要插入的元素放到最后来解决，那么以前的内存指针自然就不可用了。特别是在和 `find` 等算法在一起使用的时候，牢记这个原则：不要使用过期的 `iterator`。

(3) 当数据元素增多时，`set` 的插入和搜索速度变化如何？

如果知道 \log_2 的关系就应该彻底了解这个答案。在 `set` 中查找是使用二分查找，也就是说，如果有 16 个元素，最多需要比较 4 次就能找到结果，有 32 个元素，最多比较 5 次。那么有 10 000 个元素时为 $\log_{10\,000}$ ，即最多为 14 次，如果是 20000 个元素呢？最多不过 15 次。可见，当数据量增大一倍的时候，搜索次数只不过多了 1 次。当明白这个道理后，就可以安心往里面放入元素了。

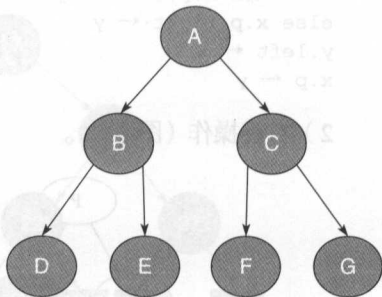


图 3-6 `set` 的结构图

3.5.2 `set` 的查增删

例 3.31 演示了 `set` 的查增删。

【例 3.31】`set` 的查增删。

```
#include <iostream>
```



```

#include <string>
#include <set>
#include <string.h>
#include <iterator>
using namespace std;
struct strLess{
    bool operator() (const char *s1, const char *s2) const {
        return strcmp(s1, s2) < 0;
    }
};

void printSet(set<int> s){
    copy(s.begin(), s.end(), ostream_iterator<int>(cout, ", "));
    /*
    set<int>::iterator iter;
    for (iter = s.begin(); iter != s.end(); iter++)
        cout<<"set["<<iter-s.begin()<<"]="<<*iter<<" "; //Error
    cout<<*iter<<" ";
    */
    cout<<endl;
}

int main(){
    /* 创建 set 对象，共 5 种方式，提示如果比较函数对象及内存分配器未出现，即表示采用的是系统默认
    方式 */
    /* 创建空的 set 对象，元素类型为 int，*/
    set<int> s1;
    /* 创建空的 set 对象，元素类型 char*，比较函数对象（即排序准则）为自定义 strLess*/
    set<const char*, strLess> s2( strLess);
    /* 利用 set 对象 s1，拷贝生成 set 对象 s2*/
    set<int> s3(s1);
    /* 用迭代区间 [&first, &last) 所指的元素，创建一个 set 对象 */
    int iArray[] = {13, 32, 19};
    set<int> s4(iArray, iArray + 3);
    /* 用迭代区间 [&first, &last) 所指的元素，及比较函数对象 strLess，创建一个 set 对象 */
    const char* szArray[] = {"hello", "dog", "bird" };
    set<const char*, strLess> s5(szArray, szArray + 3, strLess());
    /* 元素插入：
    1, 插入 value，返回 pair 配对对象，可以根据 .second 判断是否插入成功。（提示 :value 不能与
    set 容器内元素重复）
    pair<iterator, bool> insert(value)
    2, 在 pos 位置之前插入 value，返回新元素位置，但不一定能插入成功
    iterator insert(&pos, value)
    3, 将迭代区间 [&first, &last) 内所有的元素，插入到 set 容器
    void insert(&first, &last);
    */
    cout<<"s1.insert() : "<<endl;
    for (int i = 0; i < 5 ; i++)
        s1.insert(i*10);
    printSet(s1);
    cout<<"s1.insert(20).second = "<<endl;

```



```

if (s1.insert(20).second)
    cout<<"Insert OK!"<<endl;
else
    cout<<"Insert Failed!"<<endl;
cout<<"s1.insert(50).second = "<<endl;
if (s1.insert(50).second){
    cout<<"Insert OK!"<<endl;
    printSet(s1);
}else
    cout<<"Insert Failed!"<<endl;
cout<<"pair<set<int>::iterator, bool> p;\np = s1.insert(60);\nif (p.second):"<<endl;
pair<set<int>::iterator, bool> p;
p = s1.insert(60);
if (p.second){
    cout<<"Insert OK!"<<endl;
    printSet(s1);
}
else
    cout<<"Insert Failed!"<<endl;

/* 元素删除
1,size_type erase(value) 移除 set 容器内元素值为 value 的所有元素, 返回移除的元素个数
2,void erase(&pos) 移除 pos 位置上的元素, 无返回值
3,void erase(&first, &last) 移除迭代区间 [&first, &last) 内的元素, 无返回值
4,void clear(), 移除 set 容器内所有元素 */
cout<<"\ns1.erase(70) = "<<endl;
s1.erase(70);
printSet(s1);
cout<<"s1.erase(60) = "<<endl;
s1.erase(60);
printSet(s1);

cout<<"set<int>::iterator iter = s1.begin();\ns1.erase(iter) = "<<endl;
set<int>::iterator iter = s1.begin();
s1.erase(iter);
printSet(s1);

/* 元素查找
count(value) 返回 set 对象内元素值为 value 的元素个数
iterator find(value) 返回 value 所在位置, 找不到 value 将返回 end()
lower_bound(value), upper_bound(value), equal_range(value) */
cout<<"\ns1.count(10) = "<<s1.count(10)<<" , s1.count(80) = "<<s1.count(80)<<endl;
cout<<"s1.find(10) : ";
if (s1.find(10) != s1.end())
    cout<<"OK!"<<endl;
else
    cout<<"not found!"<<endl;

cout<<"s1.find(80) : ";
if (s1.find(80) != s1.end())
    cout<<"OK!"<<endl;

```

```

else
    cout<<"not found!"<<endl;
/* 其他常用函数 */
cout<<"\ns1.empty()="<<s1.empty()<<" , s1.size()="<<s1.size()<<endl;
set<int> s9;
s9.insert(100);
cout<<"s1.swap(s9) : "<<endl;
s1.swap(s9);
cout<<"s1: " <<endl;
printSet(s1);
cout<<"s9: " <<endl;
printSet(s9);
return 0;
}

```

程序的执行结果如图 3-7 所示。

例 3.31 中, 有 set 的初始化、插入、查找、删除、遍历等方法的演示, 接下来再进行详细说明。

(1) set 对象的创建方式有 5 种, 如下所述。

1) 创建空的 set 对象, 元素类型为 int:

```
set<int> s1;
```

2) 创建空的 set 对象, 元素类型 char*, 比较函数对象 (即排序准则) 为自定义 strLess:

```
set<const char*, strLess> s2( strLess);
```

3) 利用 set 对象 s1, 拷贝生成 set 对象 s2:

```
set<int> s3(s1);
```

4) 用迭代区间 [&first, &last) 所指的元素, 创建一个 set 对象:

```
int iArray[] = {13, 32, 19};
set<int> s4(iArray, iArray + 3);
```

5) 用迭代区间 [&first, &last) 所指的元素, 及比较函数对象 strLess, 创建一个 set 对象:

```
const char* szArray[] = {"hello", "dog", "bird" };
set<const char*, strLess> s5(szArray, szArray + 3, strLess() );
```

(2) 元素插入的 3 种方式, 如下所述。

1) 插入 value, 返回 pair 配对对象, 可以根据 .second 判断是否插入成功。(注意: value 不能与 set 容器内元素重复)

2) 在 pos 位置之前插入 value, 返回新元素位置, 但不一定能插入成功。

3) 将迭代区间 [&first, &last) 内所有的元素, 插入到 set 容器。

```

s1.insert() :
0, 10, 20, 30, 40,
s1.insert(20).second =
Insert Failed!
s1.insert(50).second =
Insert OK!
0, 10, 20, 30, 40, 50,
pair<set<int>::iterator, bool> p;
p = s1.insert(60);
if (p.second):
Insert OK!
0, 10, 20, 30, 40, 50, 60,

s1.erase(70) =
0, 10, 20, 30, 40, 50, 60,
s1.erase(60) =
0, 10, 20, 30, 40, 50,
set<int>::iterator iter = s1.begin();
s1.erase(iter) =
10, 20, 30, 40, 50,

s1.count(10) = 1, s1.count(80) = 0
s1.find(10) : OK!
s1.find(80) : not found!

s1.empty()=0, s1.size()=5
s1.swap(s9) :
s1:
100,
s9:
10, 20, 30, 40, 50,

```

图 3-7 例 3.31 程序的执行结果

(3) 元素删除的 4 种方式，如下所述。

- 1) `size_type erase(value)` 移除 set 容器内元素值为 value 的所有元素，返回移除的元素个数。
- 2) `void erase(&pos)` 移除 pos 位置上的元素，无返回值。
- 3) `void erase(&first, &last)` 移除迭代区间 [&first, &last) 内的元素，无返回值。
- 4) `void clear()`，移除 set 容器内所有元素。

(4) 元素查找的 2 种方式，如下所述。

- 1) `count(value)` 返回 set 对象内元素值为 value 的元素个数。
- 2) `iterator find(value)` 返回 value 所在位置，找不到 value 将返回 `end()`。

(5) 其他 set 中的常用方法。

`begin()`，返回 set 容器的第一个元素
`end()`，返回 set 容器的最后一个元素
`clear()`，删除 set 容器中的所有元素
`empty()`，判断 set 容器是否为空
`max_size()`，返回 set 容器可能包含的元素最大个数
`size()`，返回当前 set 容器中的元素个数
`rbegin()`，返回的值和 `end()` 相同
`rend()`，返回的值和 `rbegin()` 相同

3.6 本章小结

本章重点讲述了常用的几种 STL 模板：string、vector、map 和 set。基本上掌握这 4 种就能满足日常工作所需。只要会用了、用惯了，你会发现离不开它们了。

读完这一章，你会发现，不要放过任何一个看上去很简单的小编程问题，它们往往并不简单，都可以引申出很多知识点。

编译

先来看下第1章就讲到的 hello world 程序。这个程序在编译中是这样进行的，执行 `g++ helloworld.cpp` 命令编译得到了 `a.out` 文件；执行 `./a.out` 命令就可以输出“hello world.”。事实上，执行 `g++ helloworld.cpp` 命令的这个过程可以分解为4个步骤，分别是预处理、编译、汇编和链接。这就像是一个被隐藏了的过程，使用者只需用简单的命令即可完成复杂的操作步骤。

本章将和读者来一起探索下这其中的奥妙。

4.1 编译与链接

编译与链接的过程可以分解为4个步骤，分别是预处理（Prepressing）、编译（Compilation）、汇编（Assembly）和链接（Linking），具体如图4-1所示。

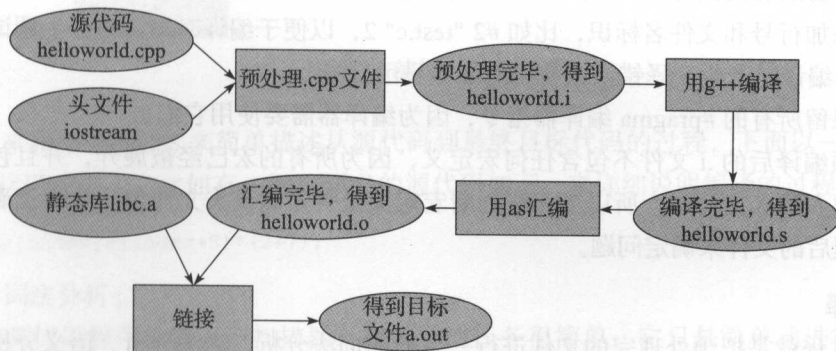


图4-1 编译与链接的过程图

1. 预处理

首先是源代码文件 `helloworld.cpp` 和相关的头文件，如 `iostream` 等被预处理器 `cpp` 预处理成一个 `.i` 文件。第一步预处理的过程相当于如下命令（`-E` 表示只进行预处理）：

```
g++ -E helloworld.cpp -o helloworld.i
```

其中：`-E` 的编译选项，意味着只执行到预编译，直接输出预编译结果。

预处理过程主要处理那些源代码文件只能够的以“`#`”开始的预编译指令。比如 `#include`、`#define` 等，主要处理规则如下所述。

（1）将所有的 `#define` 删除，并且展开所有的宏定义，如：

```
#define a b
```

对于这种伪指令，预编译所要做的是将程序中的所有 `a` 用 `b` 替换，但作为字符串常量的 `a` 则不被替换。还有 `#undef`，则将取消对某个宏的定义，使以后该串的出现不再被替换。

（2）处理所有条件预编译指令，比如 `#if`、`#ifdef`、`#elif`、`#else`、`#endif`。

这些伪指令的引入使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理。预编译程序将根据有关的文件，将那些不必要的代码过滤掉。

（3）处理 `#include` 预编译指令，将被包含的文件插入到该预编译指令的位置。注意：这个过程是递归进行的，也就是说被包含的文件可能还包含其他文件。

在头文件中一般用伪指令 `#define` 定义大量的宏（最常见的是字符常量），同时包含有各种外部符号的声明。采用头文件的目的是为了某些定义可以供多个不同的 `cpp` 源程序使用。因为在需要用到这些定义的 `cpp` 源程序中，只需加上一条 `#include` 语句即可，而不必再在此文件中将这些定义重复一遍。预编译程序将把头文件中的定义统统都加入到它所产生的输出文件中，以供编译程序对之进行处理。包含到 `cpp` 源程序中的头文件可以是系统提供的，这些头文件一般被放在 `/usr/include` 目录下。在程序中 `#include` 它们时要使用尖括号（`<>`）。另外开发人员也可以定义自己的头文件，这些文件一般与 `c` 源程序放在同一目录下，此时在 `#include` 中要用双引号（`"`）。

（4）过滤所有的注释“`//`”和“`/**/`”中的内容。

（5）添加行号和文件名标识，比如 `#2 "test.c" 2`，以便于编译时编译器产生调试用的行号信息及用于编译时产生编译错误或警告时能够显示行号。

（6）保留所有的 `#pragma` 编译器指令，因为编译器需要使用它们。

经过预编译后的 `.i` 文件不包含任何宏定义，因为所有的宏已经被展开，并且包含的文件也已经被插入到 `.i` 文件中。所以当无法判断宏定义是否正确或头文件包含是否正确时，可以查看预处理后的文件来确定问题。

2. 编译

编译过程就是把预处理完的文件进行一系列的词法分析、语法分析、语义分析以及优化后产生相应的汇编代码文件，这个过程往往是整个程序构建的核心部分，也是最复杂的部分

之一。上面的编译过程相当于如下命令：

```
g++ -S helloworld.i -o helloworld.s
```

可以使用 `vi` 直接查看 `helloworld.s` 里的汇编代码。图 4-2 只截取了汇编代码的一小部分以作展示。其中，`-S` 的编译选项，表示只执行到源代码到汇编代码的转换，输出汇编代码。注意，这里是用大写 `S`，而不是小写 `s`，因为大小写 `s` 的编译选项表示含义都是不一样的，`-s` 表示直接生成与运用 `strip` 同样效果的可执行文件（删除了所有符号信息）。

究竟编译器做了什么？从最直观的角度来讲，编译器就是将高级语言翻译成机器语言的一个工具。比如可以用 C/C++ 语言写的一个程序可以使用编译器将其翻译成机器可以执行的指令及数据。编译的过程一般分为 6 步：扫描（词法分析）、语法分析、语义分析、源代码优化、代码生成和目标代码优化，整个过程如图 4-3 所示。

```
[root@xplusrpi ~]# vi helloworld.s
1  .file "helloworld.cpp"
2  .local __ZStL8_ioint
3  .comm __ZStL8_ioint,1,1
4  .section .rodata
5  .LC0:
6  .string "hello world"
7  .text
8  .globl main
9  .type main, @function
10 main:
11 .LFB957:
12  .cfi startproc
13  .cfi personality 0x3, __gxx_personality_v0
14  pushl %ebp
15  .cfi def cfa, offset 16
16  .cfi offset 6, -16
17  movq __ksp, %rbp
18  .cfi def cfa register 6
19  movl $LC0, %esi
20  movl $ZSt4cout, %edi
```

图 4-2 helloworld.s 里的部分汇编代码

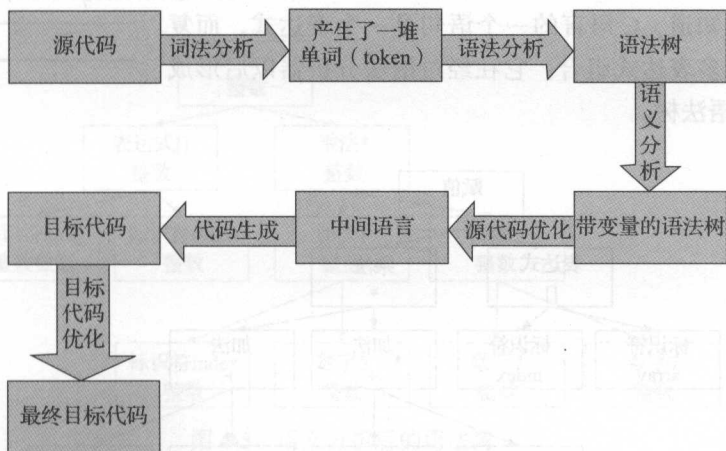


图 4-3 编译过程示意图

接下来将结合图 4-3 来简单描述从源代码到最终目标代码的过程，下面以一段很简单的 C 语义的代码为例子，比如有一行 C 语义的源代码如下，来详细说明编译的过程。

```
array[index]=(index+5)*(2+7);
```

(1) 词法分析。

首先源代码程序被输入到扫描器，扫描器的任务很简单，它只是简单地进行词法分析，运用一种类似于有限状态机的算法可以很轻松地将源代码的字符序列分割成一系列的记号。

比如示例的那行程序，总共包含了 28 个非空字符，经过扫描后，产生了 16 个记号，如表 4-1 所示。

词法分析产生的记号一般可分为如下几类：关键字、标识符、字面量（包含数字、字符串等）和特殊记号（如加号、等号）。在识别记号的同时，扫描器也完成了其他工作。比如将标识符存放到符号表中，将数字、字符串常量存放到文字表等，以备后面的步骤使用。

有一个叫 lex 的程序可以实现词法扫描，它会按照用户之前描述好的语法规则将输入的字符串分割成一个个记号。

另外对于一些有预处理的语言（如 C 语言），它的宏替换和文件包含等工作一般不归入编译器的范围而交给一个独立的预处理器。

（2）语法分析。

接下来语法分析器将对由扫描器产生的记号进行语法分析，从而产生语法树。整个分析过程采用了上下文无关文法的分析手段。简单地讲，由语法分析器生成的语法树就是以表达式为节点的树。我们知道，C 语言的一个语句是一个表达式，而复杂的表达式由很多表达式组合。它在经过语法分析器以后形成如图 4-4 所示的语法树。

表 4-1 词法分析结果

单词 (token)	类型
array	标识符
[左方括号
index	标识符
]	右方括号
=	赋值
(左圆括号
index	标识符
+	加号
5	数字
)	右圆括号
*	乘号
(左圆括号
2	数字
+	加号
7	数字
)	右圆括号

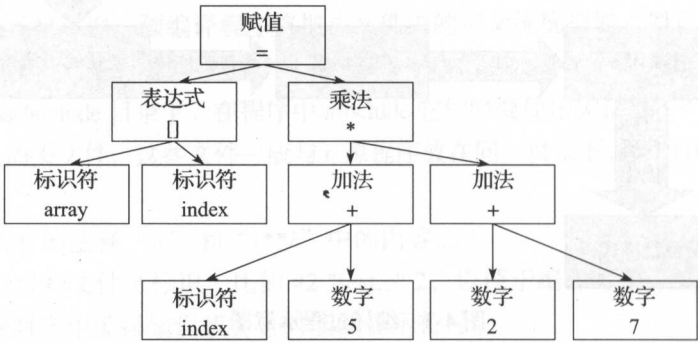


图 4-4 语法树

从图 4-4 可以看到，整个语句被看作一个赋值表达式：赋值表达式的左边是一个数组表达式；它的右边是一个乘法表达式；数组表达式又由两个符号表达式组成，等等。符号和数字是最小的表达式，它们不是由其他的表达式来组成的，所以通常为整个语法树的叶节点。在语法分析的同时，很多运算符的优先级和含义也被确定下来。比如乘法表达式的优先级比加法高，而圆括号表达式的优先级比乘法高等等。如果出现了表达式不合法，比如各种括号

不匹配、表达式中缺少操作符等，编译器就会报告语法分析阶段的相关错误。

语法分析也有一个现成的工具叫 yacc，它也像 lex 一样，可以根据用户给定的语法规则对输入的记号序列进行解析，从而构建一棵语法树。

(3) 语义分析。

语义分析是由语义分析器完成。语法分析仅仅是完成了对表达式的语法层面的分析，但是它并不了解这个语句是否真正有意义。比如对 C 语言里面两个指针做乘法运算是没有意义的，但是这个语句在语法上是合法的。编译器所能分析的语义是静态语义。所谓静态语义是指在编译期间可以确定的语义。与之对应的动态语义就是只有在运行期间才能确定的语义。

静态语义通常包括声明和类型的匹配及类型的转换等。比如当一个浮点型的表达式赋值给一个整型的表达式时，其中隐含了一个浮点型赋值给一个指针的时候，语义分析程序会发现这个类型不匹配，编译器将会报错。

动态语义一般指在运行期间出现的语义相关的问题，比如将 0 作为除数是一个运行期语义错误。

经过语义分析阶段后，整个语法树的表达式都被标识了类型，如果有些类型需要做隐式转化，语义分析程序会在语法树中插入相应的转换节点。上面描述的语法树在经过语义分析阶段以后成为了如图 4-5 所示的形式。

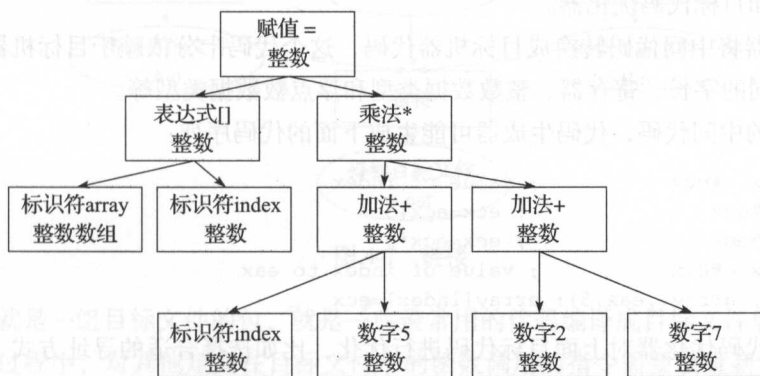


图 4-5 语义分析后的语法树

可以看到，每个表达式（包括符号和数字）都被标识了类型。在示例中几乎所有的表达式都是整型的，所以无需做转换。

语义分析器还对符号表里的符号类型也做了更新。

(4) 中间语言的生成。

现代的编译器有着很多层次的优化，往往在源代码级别会有一个优化过程。这里所描述的源码级优化器在不同编译器中可能会有不同的定义或一些其他差异。源代码优化器会在源代码级别进行优化。在示例中，可以发现， $(2+7)$ 这个表达式可以被优化掉，因为它的值在编译期间就可以被确定。

经过优化的语法树如图 4-6 所示。

可以看到 $(2+7)$ 这个表达式被优化成 9。其实直接在语法树上进行这类优化比较困难，所以源代码优化器往往将整个语法树转换成中间代码，它是语法树的顺序表示，其实它已经非常接近目标代码了。但是中间代码一般跟目标机器和运行时环境是无关的，比如不包含数据的尺寸、变量的地址和寄存器的名字等。中间代码有很多种类型，在不同的编译器中有着不同的形式，比如三址码等。

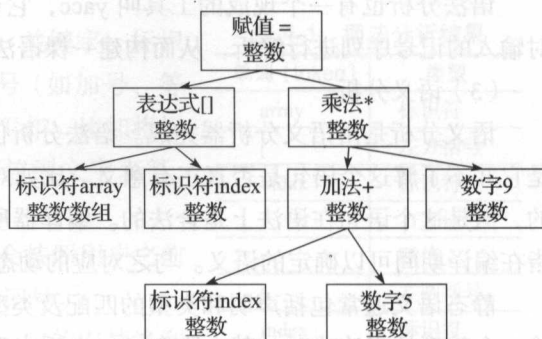


图 4-6 优化后的语法树

中间代码使得编译器可以被分成前端和后端。编译器前端负责产生机器无关的中间代码，编译器后端则负责将中间代码转换成目标机器代码。这样对于一些可以跨平台的编译器而言，它们可以针对不同的平台使用同一个前端和针对不同的机器平台的数个后端。

· (5) 目标代码的生成与优化。

源代码级优化器产生中间代码标志着下面的过程都属于编译器后端。编译器后端主要包括代码生成器和目标代码优化器。

代码生成器将中间代码转换成目标机器代码，这个代码十分依赖于目标机器，因为不同的机器有着不同的字长、寄存器、整数数据类型和浮点数数据类型等。

对于示例的中间代码，代码生成器可能生成下面的代码序列：

```

movl index, %ecx          ; value of index
addl $5, %ecx             ; ecx=ecx+5
mull $9, %ecx             ; ecx=ecx*9
movl index, %eax          ; value of index to eax
movl %ecx, array(,eax,5); array[index]=ecx
  
```

最后目标代码优化器对上面目标代码进行优化，比如选择合适的寻址方式、使用位移来代替乘法等。

经过了扫描（词法分析）、语法分析、语义分析、源代码优化、目标代码生成和目标代码优化，编译器经过这么多步骤，源代码终于被编译成了目标代码。但是这个目标代码中有一个问题：index 和 array 的地址还没有确定。如果现在把目标代码使用汇编器编译成真正能够在机器上执行的指令，那么 index 和 array 的地址从哪里得的呢？如果 index 和 array 定义在跟上面的源代码同一个编译单元里面，那么编译器可以为 index 和 array 分配空间，确定它们的地址，但如果是定义在其他的程序模块中呢？

事实上，定义其他模块的全局变量和函数在最终运行时的绝对地址都要在最终链接的时候才能确定。所以现代的编译器可以将一个源代码文件编译成一个未链接的目标文件，然后由链接器最终将这些目标文件链接起来形成可执行文件。

3. 链接

把每个源代码模块独立地编译，然后按照要将它们“组装”起来，这个组装模块的过程就是链接。链接的主要内容就是把各个模块之间相互引用的部分都处理好，使得各个模块之间能够正确的衔接。

但从原理上讲，它的工作无非就是把一些指令对其他符号地址的引用加以修正。链接过程主要包括了地址和空间分配、符号决议和重定位等这些步骤。

最基本的静态链接过程如图 4-7 所示。每个模块的源代码文件（如 .c 文件）经过编译器编译成目标文件（如 .o 文件），目标文件和库一起链接形成最终可执行文件。而最常见的库就是运行时库，它是支持程序运行的基本函数集合。

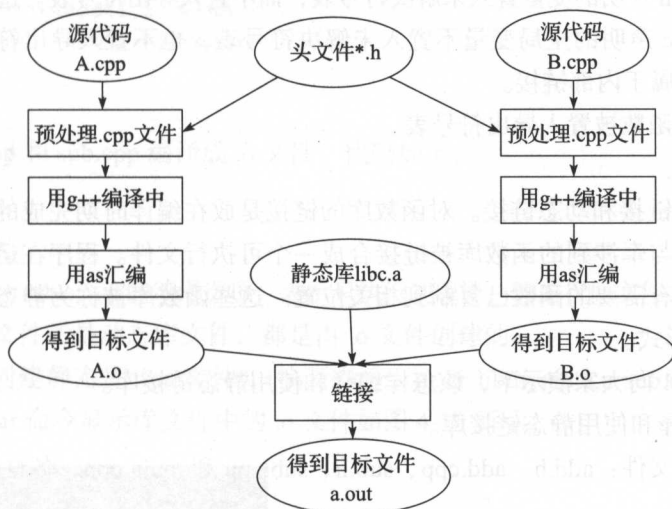


图 4-7 链接

库其实就是一组目标文件的包，就是一些最常用的代码编译成目标文件后打包存放。

在链接过程中，对其他定义在目标文件中的函数调用的指令需要被重新调整，对实用其他定义在其他目标文件的变量来说，也存在同样问题。

结合具体的 CPU 指令来深入了解这个过程。假设有个全局变量 var，它在目标文件 A 中。当要在目标文件 B 里面要访问这个全局变量，如有这么一条指令：

```
movl $0x2a, var
```

这条指令就是给这个 var 变量赋值 0x2a，相当于 C 语言中的语句 var = 42。然后编译目标文件 B，得到这条指令机器码，如图 4-8 所示。

由于在编译目标文件 B 的时候，编译器并不知道变量 var 的目标地址，所以编译器在没法确定地址的情

mov指令码

源变量

C705	00 00 00 00	2a 00 00 00
------	-------------	-------------

目标地址

图 4-8 赋值指令机器码

况下，将这条 mov 指令的目标地址设为 0，等待链接器在将目标文件 A 和 B 链接起来的时候再将其修正。假设 A 和 B 链接完成后，变量 var 的地址确定下来为 0x1000，那么链接器将会把这个指令的目标地址部分修改成 0x10000。这个地址修正的过程也叫作重定位，每个要被修正的地方叫一个重定位入口。

每个目标文件除了拥有自己的数据和二进制代码外，还提供了 3 个表：未解决符号表、导出符号表、地址重定向表，具体如下所述：①未解决符号表提供了所有在该编译单元里引用但是定义并不是在本编译单元的符号以及其出现的地址；②导出符号表提供了本编译单元具有定义，并且愿意提供给其他单元使用的符号及其地址；③地址重定向表提供了本编译单元所有对自身地址的引用的记录。

编译器将 extern 声明的变量置入未解决符号表，而不置入导出符号表。这属于外部链接。

编译器将 static 声明的全局变量不置入未解决符号表，也不置入导出符号表，因此其他单元无法使用。这属于内部链接。

普通变化及其函数被置入导出符号表。

(1) 静态链接。

链接分为静态链接和动态链接。对函数库的链接是放在编译时期完成的是静态链接。所有相关的目标文件与牵涉到的函数库被链接合成一个可执行文件。程序在运行时，与函数库再无瓜葛，因为所有需要的函数已复制到相关位置。这些函数库被称为静态库，通常文件名为“libxxx.a”的形式。

下面通过例 4.1 向大家演示下，该怎样编译和使用静态链接库。

【例 4.1】编译和使用静态链接库。

有这样的 5 个文件：add.h、add.cpp、sub.h、sub.cpp 和 main.cpp，各自代码如下所示。

add.h:

```
#ifndef _ADD_H_
#define _ADD_H_
int add(int a, int b);
#endif
```

add.cpp:

```
#include "add.h"
int add(int a, int b){
    return a+b;
}
```

sub.h:

```
#ifndef _SUB_H_
#define _SUB_H_
int sub(int a, int b);
#endif
```


sub.cpp:

```
#include "sub.h"
int sub(int a,int b){
    return a-b;
}
```

main.cpp:

```
#include "add.h"
#include "sub.h"
#include "iostream"
using namespace std;
int main(){
    cout<<"1+2="<<add(1,2)<<endl;
    cout<<"1-2="<<sub(1,2)<<endl;
    return 0;
}
```

1) 先将 add.cpp 和 sub.cpp 编译成 .o 文件, 代码如下:

```
g++ -c add.cpp
g++ -c sub.cpp
```

生成的文件 add.o sub.o, -c 的编译选项, 表示只执行到编译, 输出目标文件, 如图 4-9 所示。无论是静态库文件还是动态库文件, 都是由 .o 文件创建的。

2) 由 .o 文件创建静态库 (.a 文件), 执行命令: ar cr libmymath.a sub.o add.o, 会生成 libmymath.a 文件, ar 命令显示库文件中的 .o 文件如图 4-10 所示。

```
[root@linux chapter04]# g++ -c add.cpp
[root@linux chapter04]# ls
add.cpp  add.h  add.o  main.cpp  sub.cpp  sub.h
[root@linux chapter04]# g++ -c sub.cpp
[root@linux chapter04]# ls
add.cpp  add.h  add.o  main.cpp  sub.cpp  sub.h  sub.o
[root@linux chapter04]#
```

图 4-9 例 4.1 编译生成 .o 文件

```
[root@linux chapter04]# ar tv libmymath.a
rw-r--r-- 0/0 1320 Oct 17 22:15 2015 add.o
rw-r--r-- 0/0 1320 Oct 17 22:16 2015 sub.o
```

图 4-10 例 4.1 ar 命令显示库文件中的 .o 文件

库文件的命名规范是以 lib 开头 (前缀), 紧接着是静态库名, 以 .a 为后缀名。

ar 命令的 r 选项: 在库中插入模块 (替换)。当插入的模块名已经在库中存在, 则替换同名的模块。如果若干模块中有一个模块在库中不存在, ar 会显示一个错误消息, 并不替换其他同名模块。默认的情况下, 新的成员增加在库的结尾处, 可以使用其他任选项来改变增加的位置。

ar 命令的 c 选项: 创建一个库。不管库是否存在, 都将创建。

顺便介绍下 ar 的 tv 参数, ar tv libxxx.a, 可以显示库文件中有哪些目标文件, 显示文件名、时间、大小等详细信息。

3) 在程序中使用静态库, 执行命令 g++ -o main main.cpp -L. -lmymath, 会生成 main 文件。

静态库制作完了，要使用它内部的函数，只需要在使用到这些公用函数的源程序中包含这些公用函数的原型声明，然后再用 `g++` 命令生成目标文件时指明静态库名（是 `mymath` 而不是 `libmymath.a`），`g++` 将会从静态库中将公用函数连接到目标文件中。注意，`g++` 会在静态库名前加上前缀 `lib`，然后追加扩展名 `.a` 得到的静态库文件名来查找静态库文件。在程序 `main.cpp` 中就包含了静态库的头文件 `add.h` 和 `sub.h`，然后在主程序 `main` 中直接调用公用函数 `add()` 和 `sub()` 即可。

4) 生成目标程序 `main`，执行 `main` 文件，输出的结果如下所示：

```
1+2=3
1-2=-1
```

(2) 动态链接。

除了静态链接，也可以把对一些库函数的链接载入推迟到程序运行时期（`runtime`），这就是动态链接库（`dynamic link library`）技术。动态库文件名命名规范和静态库文件名命名规范类似，也是在动态库名增加前缀 `lib`，但其文件扩展名为 `.so`。例如：将创建的动态库名为 `mymath` 后，则动态库文件名就是 `libmymath.so`。下面通过实际的例子向大家演示下，该怎样编译和使用动态链接库。

还是采用上面例 4.1 中的 `add.h`、`add.cpp`、`sub.h`、`sub.cpp` 和 `main.cpp`，文件内容一致，键入以下命令得到动态库文件 `libmamath.so`。可以用以下命令：

```
g++ -fPIC -o add.o -c add.cpp
g++ -fPIC -o sub.o -c sub.cpp
g++ -shared -o libmymath.so add.o sub.o
```

也可以直接一条命令搞定：

```
g++ -fPIC -shared -o libmymath.so add.cpp sub.cpp
```

逐步生成动态库文件的过程如图 4-11 所示。

常用的编译参数对应的功能如下所述。

-fPIC：表示编译为位置独立的代码。不用此选项的话编译后的代码是位置相关的，所以动态载入时是通过代码复制的方式来满足不同进程的需要，而不能达到真正代码段共享的目的。

-Lpath：表示在 `path` 目录中搜索库文件，如 `-L.` 则表示在当前目录。

-Ipath：表示在 `path` 目录中搜索头文件。

-ltest：编译器查找动态链接库时有隐含的命名规则，即在给出的名字前面加 `lib`，后面加上 `.so` 来确定库的名称。

```
[root@linux chapter04]# ls
add.cpp add.h main.cpp sub.cpp sub.h
[root@linux chapter04]# g++ -fPIC -o add.o -c add.cpp
[root@linux chapter04]# ls
add.cpp add.h add.o main.cpp sub.cpp sub.h
[root@linux chapter04]# g++ -fPIC -o sub.o -c sub.cpp
[root@linux chapter04]# ls
add.cpp add.h add.o main.cpp sub.cpp sub.h sub.o
[root@linux chapter04]# g++ -shared -o libmymath.so add.o sub.o
[root@linux chapter04]# ls
add.cpp add.o main.cpp sub.h
add.h libmymath.so sub.cpp sub.o
[root@linux chapter04]#
```

图 4-11 逐步生成动态库文件

在程序中隐式使用动态库和使用静态库完全一样，也是在使用到这些公用函数的源程序中包含这些公用函数的原型声明，然后用 g++ 命令对生成目标文件时指明的动态库名进行编译。先运行 g++ 命令生成目标文件，再运行它看看结果，代码如下：

```
g++ -o main main.cpp -L. -lmymath
./main
```

链接时是正常的，但是执行的时候则报错了，提示 ./main: error while loading shared libraries: libmymath.so: cannot open shared object file: No such file or directory，如图 4-12 所示。

错误提示说明是找不到动态库文件 libmymath.so。程序在运行时，会在 /usr/lib 和 /lib 等目录中查找需要的动态库文件。若找到，则载入动态库，否则将提示类似上述错误而终止程序运行。

```
[root@linux chapter04]# g++ -shared -o libmymath.so add.o sub.o
[root@linux chapter04]# ls
add.cpp  add.o      main.cpp  sub.h
add.h    libmymath.so  sub.cpp  sub.o
[root@linux chapter04]# g++ -o main main.cpp -L. -lmymath
[root@linux chapter04]# ./main
./main: error while loading shared libraries: libmymath.so: cannot
open shared object file: No such file or directory
[root@linux chapter04]#
```

图 4-12 找不到动态库时会提示的错误

动态库的搜索路径搜索的先后顺序是：

①编译目标代码时指定的动态库搜索路径；②环境变量 LD_LIBRARY_PATH 指定的动态库搜索路径；③配置文件 /etc/ld.so.conf 中指定的动态库搜索路径；即只需在该文件中追加一行库所在的完整路径如 "/root/test/conf/lib" 即可，然后 ldconfig 是修改生效；④默认的动态库搜索路径 /lib；⑤默认的动态库搜索路径 /usr/lib。

为此解决步骤为：①将文件 libmymath.so 复制到目录 /usr/lib 中：cp libmymath.so /usr/lib/；②修改环境变量 LD_LIBRARY_PATH，具体的命令是：

```
export LD_LIBRARY_PATH=/usr/lib:$LD_LIBRARY_PATH
sudo ldconfig
```

修改了环境变量后，就可以顺利执行了，如图 4-13 所示。

(3) 动态库与静态库重名问题。

还是采用上面的 add.h、add.cpp、sub.h、sub.cpp 和 main.cpp，文件内容一致，执行以下命令：

```
[root@linux chapter04]# g++ -o main main.cpp -L. -lmymath
[root@linux chapter04]# ./main
./main: error while loading shared libraries: libmymath.so: cannot
open shared object file: No such file or directory
[root@linux chapter04]# cp libmymath.so /usr/lib/
[root@linux chapter04]# export LD_LIBRARY_PATH=/usr/lib:$LD_LIBR
Y_PATH
[root@linux chapter04]# sudo ldconfig
[root@linux chapter04]# ./main
1+2=3
1-2=-1
[root@linux chapter04]#
```

图 4-13 修改动态库的环境变量

```
g++ -c add.cpp
g++ -c sub.cpp
ar cr libmymath.a add.o sub.o
g++ -shared -fPIC -o libmymath.so add.cpp sub.cpp
```

程序执行完后会同时生成动态库与静态库，如图 4-14 所示。

现在目录有 2 个同名的库文件（动态库文件和静态库文件同名）：libmymath.a 和 libmymath.so。编译运行程序，提示 ./main: error while loading shared libraries: libmymath.so: cannot open shared object file: No such file or directory。也就是动态库文件和静态库文件同名的时

候，编译器会先到 path 目录下搜索 libxxx.so 文件，如果没有找到，则继续搜索 libxxx.a（静态库）。

```
[root@linux chapter04]# g++ -c add.cpp
[root@linux chapter04]# g++ -c sub.cpp
[root@linux chapter04]# ar cr libmymath.a add.o sub.o
[root@linux chapter04]# g++ -shared -fPIC -o libmymath.so add.cpp
sub.cpp
[root@linux chapter04]# ls
add.cpp  add.o      libmymath.so  main.cpp  sub.h
add.h    libmymath.a  main          sub.cpp   sub.o
[root@linux chapter04]#
```

图 4-14 同时生成动态库和静态库

（4）静态链接库、动态链接库各自的特点。

1）动态链接库有利于进程间资源共享。

当某个程序在运行中要调用某个动态链接库函数的时候，操作系统首先会查看所有正在运行的程序，看在内存里是否已有此库函数的拷贝了。如果有，则让其共享那一个拷贝；如果没有时才链接载入。这样的模式虽然会带来一些“动态链接”额外的开销，却大大节省了系统的内存资源。C 语言的标准库就是动态链接库，也就是说系统中所有运行的程序共享着同一个 C 语言标准库的代码段。而静态链接库则不同，如果系统中多个程序都要调用某个静态链接库函数时，则每个程序都要将这个库函数拷贝到自己的代码段中，这显然将占有更大的内存资源。

2）将一些程序升级变得简单。用静态库，如果库发生变化，使用库的程序要重新编译；使用动态库，只要动态库提供给该程序的接口没变，只要重新用新生成的动态库替换原来就可以了。

3）甚至可以真正做到链接载入完全由程序员在程序代码中控制。

程序员在编写程序的时候，可以明确的指明什么时候或者什么情况下，链接载入哪个动态链接库函数。你可以有一个相当大的软件，但每次运行的时候，由于不同的操作需求，只有一小部分程序被载入内存。但若所有的函数本着“有需求才调入”的原则，则可以大大节省系统资源。比如现在的软件通常都能打开若干种不同类型的文件，这些读写操作通常都用动态链接库来实现，在一次运行当中，一般只有一种类型的文件将会被打开。所以直到程序知道文件的类型以后再载入相应的读写函数，而不是一开始就将所有的读写函数都载入，然后才发觉在整个程序中根本没有用到它们造成低效。

4）由于静态库在编译的时候，就将库函数装载到程序中去了，而动态库函数必须在运行的时候才被装载，所以程序在执行的时候，用静态库速度更快些。

4. g++ 与 gcc 的区别

在编译 C/C++ 代码的时候，有人用 gcc，有人用 g++，于是各种说法都来了，譬如 C 代码用 gcc，而 c++ 代码用 g++，或者说编译用 gcc，链接用 g++，一时也不知哪个说法正确，如果再遇上个 extern"C"，分歧就更多了。本节就简单讲讲这二者的区别与联系。

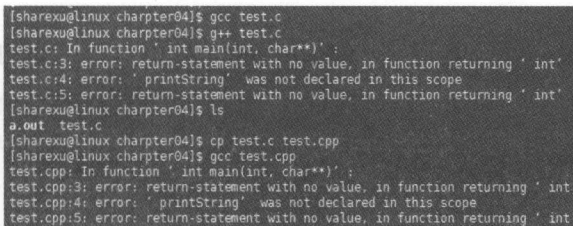
（1）误区一：gcc 只能编译 C 代码，g++ 只能编译 c++ 代码。事实上，两者都可以，但

是请注意，以下几点。

1) 后缀为 .c 的，gcc 把它当作是 C 程序，而 g++ 当作是 C++ 程序；后缀为 .cpp 的，两者都会认为是 C++ 程序，注意，虽然 C++ 是 C 的超集，但是两者对语法的要求是有区别的，例如：

```
#include<stdio.h>
int main(int argc,char* argv[]){
    if(argv==0)return;
    printString(argv);
    return;
}
int printString(char* string){
    sprintf(string,"This is a test.\n");
}
```

如果按照 C 语言的语法规则，是没有问题的，但一旦把后缀改为 cpp，立刻报两个错：“printString 未定义”和“return-statement with no value”，如图 4-15 所示。



```
[sharexu@linux chapter04]$ gcc test.c
[sharexu@linux chapter04]$ g++ test.c
test.c: In function 'int main(int, char**)':
test.c:3: error: return-statement with no value, in function returning 'int'
test.c:4: error: 'printString' was not declared in this scope
test.c:5: error: return-statement with no value, in function returning 'int'
[sharexu@linux chapter04]$ ls
a.out test.c
[sharexu@linux chapter04]$ cp test.c test.cpp
[sharexu@linux chapter04]$ gcc test.cpp
test.cpp: In function 'int main(int, char**)':
test.cpp:3: error: return-statement with no value, in function returning 'int'
test.cpp:4: error: 'printString' was not declared in this scope
test.cpp:5: error: return-statement with no value, in function returning 'int'
```

图 4-15 按 C 的语法规则和按 C++ 语法规则编译的不同结果

可见 C++ 的语法规则更加严谨一些。

2) 编译阶段，g++ 会调用 gcc，对于 C++ 代码，两者是等价的；但是因为 gcc 命令不能自动和 C++ 程序使用的库链接，所以通常用 g++ 来完成链接，为了统一起见，干脆编译/链接统统用 g++ 了，这就给人一种错觉，好像 cpp 程序只能用 g++ 似的。

(2) 误区二：gcc 不会定义 __cplusplus 宏，而 g++ 会。

实际上，这个宏只是标志着编译器将会把代码按 C 还是 C++ 语法来解释，如上所述，如果后缀为 .c，并且采用 gcc 编译器，则该宏就是未定义的，否则就是已定义。

(3) 误区三：编译只能用 gcc，链接只能用 g++。

严格来说，这句话不算错误，但是它混淆了概念，应该这样说：编译可以用 gcc/g++，而链接可以用 g++ 或者 gcc-lstdc++。因为 gcc 命令不能自动和 C++ 程序使用的库链接，所以通常使用 g++ 来完成链接。但在编译阶段，g++ 会自动调用 gcc，二者等价。

(4) 误区四：extern"C" 与 gcc/g++ 有关系。

实际上并无关系，无论是 gcc 还是 g++，用 extern"C" 时，都是以 C 的命名方式来为 symbol 命名；否则，都以 C++ 方式命名。

【例 4.2】 使用 extern"C" 编译程序。

me.h 的代码是：

```
extern "C" void CppPrintf (void);
```

me.cpp 的代码是：

```
#include<iostream>
#include"me.h"
using namespace std;
void CppPrintf(void){
    cout<<"Hello"<<endl;
}
```

test.cpp 的代码是：

```
#include"me.h"
int main(){
    CppPrintf();
    return 0;
}
```

先给 me.h 加上 extern"C"，看用 gcc 和 g++ 命名有什么不同。执行以下命令：

```
g++ -S me.cpp
less me.s
```

可以看到结果是：

```
.globl CppPrintf
.type CppPrintf, @function
```

执行以下命令：

```
gcc -S me.cpp
less me.s
```

可以看到结果是：

```
.globl CppPrintf
.type CppPrintf, @function
```

也就是说，加上了 extern"C" 后，CppPrintf 这个函数用 g++ 和 gcc 编译得到的函数命名是一样的，都是以 C 的命名方式。再把 me.h 的 extern"C" 去掉。执行以下命令：

```
g++ -S me.cpp
less me.s
```

可以看到结果是：

```
.globl _Z9CppPrintfv
.type _Z9CppPrintfv, @function
```


执行以下命令：

```
gcc -S me.cpp
less me.s
```

可以看到结果是：

```
.globl _Z9CppPrintfv
.type _Z9CppPrintfv, @function
```

也就算说，去掉了 `extern"C"` 后，`CppPrintf` 函数用 `g++` 和 `gcc` 编译得到的函数命名是一样的，都是以 C++ 的命名方式。可见 `extern"C"` 与采用 `gcc/g++` 并无关系。

4.2 makefile 的撰写

一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，如何更高效地编译整个工程，需要用到 `makefile` 和 `make` 命令工具。`makefile` 中会定义一系列的规则，指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作。

`makefile` 带来的好处就是“自动化编译”，一旦写好，只需要一个 `make` 命令，整个工程完全自动编译，极大地提高了软件开发的效率。`make` 命令是一个命令工具，是一个解释 `makefile` 中指令的命令工具，一般来说，大多数的 IDE 都有这个命令，比如：Delphi 的 `make` 命令，Visual C++ 的 `nmake` 命令，Linux 下 GNU 的 `make` 命令等。可见，利用 `makefile` 进行编译，已成为了一种在工程方面的常见编译方法。

会不会写 `makefile`，可以从一个侧面说明一个人是否具备完成大型工程的能力。因为，`makefile` 关系到了整个工程的编译规则。

`makefile` 就像一个 `shell` 脚本一样，其中也可以执行操作系统的命令。

下面将用一个示例来说明 `makefile` 的书写规则。

【例 4.3】 `makefile` 书写规则举例。

准备 3 个文件：`file1.h`, `file1.cpp`, `file2.cpp`。

`file1.h` 的代码是：

```
#ifndef FILE1_H_
#define FILE1_H_
#ifdef __cplusplus
extern "C" {
    #endif
    void File1Print();
    #ifdef __cplusplus
    }
    #endif
#endif
```


file1.cpp 的代码是：

```
#include <iostream>
#include "file1.h"
using namespace std;
void File1Print(){
    cout<<"Print file1*****"<<endl;
}
```

file2.cpp 的代码是：

```
#include <iostream>
#include "file1.h"
using namespace std;
int main(){
    cout<<"Print file2*****"<<endl;
    File1Print();
    return 0;
}
```

makefile 文件是：

```
helloworld:file1.o file2.o
    g++ file1.o file2.o -o helloworld

file2.o:file2.cpp
    g++ -c file2.cpp -o file2.o

file1.o:file1.cpp file1.h
    g++ -c file1.cpp -o file1.o

clean:
    rm -rf *.o helloworld
```

可见，一个 makefile 主要含有一系列的规则，如下所示：

```
A: B
(tab)<command>
(tab)<command>
```

每个命令行前都必须有 tab 符号。上面的 makefile 文件目的就是要编译一个 helloworld 的可执行文件，接下来一句一句来解释。

helloworld 依赖 file1.o file2.o 两个目标文件：

```
helloworld : file1.o file2.o
```

编译出 helloworld 可执行文件，-o 后面加你指定的目标文件名：

```
g++ file1.o file2.o -o helloworld
```

file2.o 依赖 file2.cpp 文件：

```
file2.o:file2.cpp
```

下面是编译出 file2.o 文件。-c 表示 g++ 只把给它的文件编译成目标文件，用源码文件的文件名命名但把其后缀由“.c”或“.cc”或“.cpp”变成“.o”。在这句中，可以省略-o file2.o，编译器默认生成 file2.o 文件，这就是 -c 的作用。

```
g++ -c file2.cpp -o file2.o
```

编译出 file1.o 文件：

```
file1.o:file1.cpp file1.h
g++ -c file1.cpp -o file1.o
```

当用户键入 "make clean" 命令时，会删除 *.o 和 helloworld 文件。写好 makefile 文件，在命令行中直接键入 make 命令，就会执行 makefile 中的内容了：

```
clean:
    rm -rf *.o helloworld
```

到这步大家都能轻松编译一个 helloworld 程序了。再上一层楼，下面来学学怎么使用变量。可以看到，上面有很多 g++，其实可以把它写到一个变量里。一个使用变量的 makefile 如例 4.4 所示。

【例 4.4】 在 makefile 中使用变量。

```
OBJS = file1.o file2.o
XX = g++
CFLAGS = -Wall -O -g

helloworld : $(OBJS)
    $(XX) $(OBJS) -o helloworld

file2.o : file2.cpp file1.h
    $(XX) $(CFLAGS) -c file2.cpp -o file2.o

file1.o : file1.cpp file1.h
    $(XX) $(CFLAGS) -c file1.cpp -o file1.o

clean:
    rm -rf *.o helloworld
```

执行 make 命令后提示：

```
g++ -Wall -O -g -c file1.cpp -o file1.o
g++ -Wall -O -g -c file2.cpp -o file2.o
g++ file1.o file2.o -o helloworld
```

并生成了 helloworld 文件。

这里应用到了变量。要设定一个变量，只要在一行的前端写下这个变量的名字，后面跟

一个“=”号，后面跟要设定的这个变量的值即可。以后要引用这个变量，只写一个“\$”符号，后面是在括号里的变量名即可。

CFLAGS = -Wall -O -g：配置编译器设置，并把它赋值给 CFLAGS 变量，其中每个部分含义为：① -Wall：输出所有的警告信息；② -O：编译时进行优化；③ -g：表示编译 debug 版本。

这样写的 makefile 文件比较简单，但很容易就会发现其缺点，那就是要列出所有的 c 文件。如果添加一个 c 文件，那就需要修改一次 makefile 文件，这在实际项目开发中还是比较麻烦的。

其实这就是编程序，只不过用的语言不同而已。接下来，将介绍如何在 makefile 里使用函数。

【例 4.5】在 makefile 里使用函数。

```
CC = gcc
XX = g++
CFLAGS = -Wall -O -g
TARGET = helloworld

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

%.o: %.cpp
    $(XX) $(CFLAGS) -c $< -o $@

SOURCES = $(wildcard *.c *.cpp)
OBJS = $(patsubst %.c,%.o,$(patsubst %.cpp,%.o,$(SOURCES)))

$(TARGET) : $(OBJS)
    $(XX) $(OBJS) -o $(TARGET)

clean:
    rm -rf *.o helloworld
```

执行 make 命令输出：

```
g++ -Wall -O -g -c file1.cpp -o file1.o
g++ -Wall -O -g -c file2.cpp -o file2.o
g++ file1.o file2.o -o helloworld
```

在 makefile 规则中，通配符会被自动展开。但在变量的定义和函数引用时，通配符将失效。这种情况下如果需要通配符有效，就需要使用函数 wildcard，它的用法是：

\$(wildcard PATTERN...) 在 makefile 中，它被展开为已经存在的、使用空格分开的、匹配此模式的所有文件列表。如果不存在任何符合此模式的文件，函数会忽略模式字符并返回空。需要注意的是：这种情况下的规则中通配符的展开和上一小节匹配通配符是有区别的。下面这一行表示产生一个所有以 .c、.cpp 结尾的文件的列表，然后存入变量 SOURCES 里。

```
SOURCES = $(wildcard *.c *.cpp)
```

patsubst 函数，用于匹配替换，有 3 个参数。第一个是一个需要匹配的式样，第二个表

示用什么来替换它，第三个是一个需要被处理的由空格分隔的列表，比如：

```
(patsubst %.c,%.o,$(dir) )
```

是指用 `patsubst` 把 `$(dir)` 中的变量符合后缀是 `.c` 的全部替换成 `.o`。而下面这一行代码，则表示把文件列表中所有的 `.c`、`.cpp` 字符变成 `.o`，形成一个新的文件列表，然后存入 `OBJS` 变量中。

```
OBJS = $(patsubst %.c,%.o,$(patsubst %.cpp,%.o,$(SOURCES)))
```

这几句命令表示把所有的 `.c`、`.cpp` 文件编译成 `.o` 文件。

```
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
%.o: %.cpp
    $(XX) $(CFLAGS) -c $< -o $@
```

这里有 3 个比较有用的内部变量：① `$@` 扩展成当前规则的目的文件名；② `$<` 扩展成依靠列表中的第一个依靠文件；③而 `^` 扩展成整个依靠的列表（除掉了里面所有重复的文件名）。

到这里就已经能够编写一个比较简单、通用的 `makefile` 文件了，需要注意的是上面所有的例子都假定所有的文件都在同一个目录下，不包含子目录。

4.3 目标文件

前面已经多次提到目标文件，究竟目标文件是什么？ELF 是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储的标准文件格式。ELF 标准的目的是为软件开发人员提供一组二进制接口定义，这些接口可以延伸到多种操作环境中，从而减少重新编码、编译程序的需要。

UNIX 最早的可执行文件格式为 `a.out` 格式，它的设计非常地简单，以至于后来当共享库这个概念出现的时候，`a.out` 格式就变得捉襟见肘了。于是人们设计了 COFF 格式标准来解决这些问题，这个设计非常通用。而 ELF 正是从 COFF 继承来的。

COFF 的主要贡献是在目标文件里面引入了“段”的机制，不同的目标文件可以拥有不同数量及不同类型的“段”。另外，它还定义了调试数据格式。ELF 格式比 COFF 更具可扩展性与灵活性，被用来取代 COFF。现在，ELF 的使用已经非常广泛了。

4.3.1 ELF 的文件类型

目标文件有 3 种类型，如下所述。

(1) 可重定位的目标文件。

这是由编译器汇编生成的 `.o` 文件，链接器拿一个或一些可重定位的目标文件作为输入，

经链接处理后，生成一个可执行的目标文件或者一个可被共享的对象文件（.so 文件）。可以使用 ar 工具将众多的 .o 文件归档 (archive) 成 .a 静态库文件。

(2) 可执行的目标文件。

文本编辑器 vi、调式用的工具 gdb、播放 mp3 歌曲的软件 mplayer 等都是可执行的目标文件。在 Linux 系统里面，存在两种可执行的文件，除了这里说的可执行的目标文件，另外一种就是可执行的脚本（如 shell 脚本）文件。注意这些脚本不是可执行的目标文件，它们只是文本文件，执行这些脚本所用的解释器才是可执行的，比如 bash shell 程序。此类文件规定了如何利用 exec() 创建一个程序的进程映像。

(3) 可被共享的目标文件。

这些就是所谓的动态库文件，也即 .so 文件。如果拿前面的静态库来生成可执行程序，那每个生成的可执行程序中都会有一份库代码的拷贝。如果在磁盘中存储这些可执行程序，那就会占用额外的磁盘空间；另外如果把它们放到 Linux 系统上一起运行，也会浪费掉宝贵的物理内存。如果将静态库换成动态库，那么这些问题都不会出现。动态库在发挥作用的过程中，必须经过两个步骤：①链接器拿它和其他可重定位的文件（.o 文件）以及其他 .so 文件作为输入，经链接处理后，生成另外的可共享的目标文件（.so 文件）或者可执行的目标文件；②在运行时，动态链接器拿它和一个可执行的目标文件以及另外一些可共享的目标文件（.so）来一起处理，在 Linux 系统里面创建一个进程映像。

4.3.2 链接视图下的 ELF 内容

有两种视图可以来说明 ELF 的组成格式，具体见表 4-2。

表 4-2 中展示了 ELF 的链接视图和执行视图。

为什么会有左右两个很类似的图来说明 ELF 的组成格式？这是因为 ELF 格式需要使用在两种场合：①组成不同的可重定位文件，以参与可执行文件或者可被共享的对象文件的链接构建②组成可执行文件或者可被共享的对象文件，以在运行时内存中进程映像的构建。

所以，基本上，图中左边的部分表示的是可重定位文件的格式；而右边部分表示的则是可执行文件以及可被共享的对象文件的格式。ELF 文件头被固定地放在不同类对象文件的最前面。因此，我们可以用 file 命令来看文件是属于哪种 ELF 文件。

继续拿前面例 4.2 中的 add.h,add.cpp,sub.h,sub.cpp,main.cpp 编译得到的 add.o、sub.o、libmymath.so 和 main4 个文件来看看。执行命令 file add.o sub.o libmymath.so main 得到的结果如图 4-16 所示。

表 4-2 ELF 对象文件组成

链接视图	执行视图
ELF 头部	ELF 头部
程序头部表 (可选)	程序头部表
节区 1	节区 1
...	...
节区 n	节区 n
...	...
...	...
节区头部表	节区头部表 (可选)


```
[sharexu@linux_elf_example]$ ls
add.cpp add.h add.o libmymath.a libmymath.so main main.cpp sub.cpp sub.h sub.o
[sharexu@linux_elf_example]$ file add.o sub.o libmymath.so main
add.o:      ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
sub.o:      ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
libmymath.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, not stripped
main:       ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs),
for GNU/Linux 2.6.18, not stripped
```

图 4-16 执行 file 命令获取文件属性

结果展示了，add.o 和 sub.o 都是可重定位文件，libmymath.so 是可被共享文件，main 是可执行文件。

1. ELF 头部

ELF 里面的内容，除了 file 命令所显示出来的那些之外，更重要的是包含另外一些数据，用于描述 ELF 文件中 ELF 文件头之外的内容。可以使用 readelf 工具来读出整个 ELF 文件头的内容，比如执行 readelf -h add.o 命令得到结果如图 4-17 所示。

(1) readelf -h add.o 命令是显示 add.o 的 ELF Header 的文件头信息（就是 ELF 文件开始的前 52 Byte），如图 4-17 所示。注意，readelf 不支持显示 archive 文档（如 .a 文件）。这个输出结果能反映出很多东西。比如：

```
[sharexu@linux_elf_example]$ readelf -h add.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:       ELF64
  Data:        2's complement, little endian
  Version:     1 (current)
  OS/ABI:      UNIX - System V
  ABI Version: 0
  Type:        REL (Relocatable file)
  Machine:     Advanced Micro Devices X86-64
  Version:     0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 288 (bytes into file)
  Flags:       0x0
  Size of this header: 64 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 64 (bytes)
  Number of section headers: 11
  Section header string table index: 8
```

图 4-17 执行 readelf 命令获取文件 ELF 头部

1) 这个 add.o 的进入点是 0x0(e_entry)。

Entry point address: 0x0

这表明可重定位文件（.o 文件）不会有程序进入点。所谓程序进入点是指当程序真正执行起来的时候，其第一条要运行的指令的地址。因为可重定位文件只是供再链接而已，所以它不存在进入点；而可执行文件 test 和动态库 .so 都存在所谓的进入点。

由图 4-18 可见，.so 文件和可执行文件的程序执行进入地址不为 0。实际上，可执行文件的 e_entry 指向 C 库中的 _start，而动态库 .so 中的进入点指向 call_gmon_start。

2) 这个 add.o 文件包含有 11 个 section（节区），但 program headers 的数量为 0。而可执行文件 main 和可被共享文件 libmymath.so 里的 program headers 则不为 0。

(2) 那什么是所谓 section 呢？可以说，section 是在 ELF 文件里头，用以装载内容数据的最小容器。在 ELF 文件里面，每一个 section 内都装载了性质属性都一样的内容，比如下几种情况。

1) text section 里装载了可执行代码。

2) data section 里面装载了被初始化的数据。

3) bss section 里面装载了未被初始化的数据。


```
[sharexu@linux elf_example]$ readelf -h libmymath.so
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:   ELF64
  Data:    2's complement, little endian
  Version: 1 (current)
  OS/ABI:  UNIX - System V
  ABI Version:
  Type:    DYN (Shared object file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x520
  Start of program headers: 64 (bytes into file)
  Start of section headers: 2648 (bytes into file)
  Flags:    0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 6
  Size of section headers: 64 (bytes)
  Number of section headers: 28
  Section header string table index: 25
[sharexu@linux elf_example]$ readelf -h main
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:   ELF64
  Data:    2's complement, little endian
  Version: 1 (current)
  OS/ABI:  UNIX - System V
  ABI Version:
  Type:    EXEC (Executable file)
  Machine: Advanced Micro Devices X86-64
  Version: 0x1
  Entry point address: 0x4008d0
  Start of program headers: 64 (bytes into file)
  Start of section headers: 5080 (bytes into file)
  Flags:    0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 8
  Size of section headers: 64 (bytes)
  Number of section headers: 30
  Section header string table index: 27
[sharexu@linux elf_example]$
```

图 4-18 用 readelf 查看动态库和可执行文件的 ELF 头部

4) 以 .rec 打头的 sections 里面装载了重定位条目。

5) symtab 或者 dynsym section 里面装载了符号信息。

6) strtab 或者 dynstr section 里面装载了字符串信息。

7) 其他还有为满足不同目的所设置的 section，比如满足调试的目的、满足动态链接与加载的目的等。

2. ELF section 表的总体预览

可以利用 readelf 工具来查看可重定位对象文件 add.o 的 section 表内容，执行 readelf -S add.o 命令得到结果如图 4-19 所示。

图 4-19 显示了 add.o 中包含的所有 11 个 section 的内容。因为 add.o 仅仅是参与链接的可重定位文件，而不参与最后进程映像的构建，所以 Address 的值为 0。后面会看到可执行文件以及动态库文件中大部分 sections 的这一字段都是有某些特殊取值的。Offset 表示了该 section 离开文件头部位置的距离；Size 表示 section 的字节大小；EntSize 只对某些形式的 sections 有意义。如符号表 .symtab section，其内部包含了一个表格，表格的每一个条目都是特定长度的，此时就表示条目的长度为 10。Align 是地址对齐要求；另外剩下的两列 Link 和 Info，它们中记录的是 section head table 中的条目索引，这就意味着，从这两个字段出发，可以找到对应的另外两个 section，其具体的含义解释依据不同种类的 section 而不同，这里不做介绍。

```
[sharexu@linux elf_example]$ readelf -S add.o
There are 11 section headers, starting at offset 0x120:

Section Headers:
 [Nr] Name              Type              Address           Offset
     Size             EntSize          Flags            Link Info  Align
 [ 0] 0000000000000000 NULL             0000000000000000 0 0 0
 [ 1] .text               PROGBITS          0000000000000000 00000040
     0000000000000015 AX 0 0 4
 [ 2] .data               PROGBITS          0000000000000000 00000058
     0000000000000005 WA 0 0 4
 [ 3] .bss                NOBITS           0000000000000000 00000058
     0000000000000000 WA 0 0 4
 [ 4] .comment            PROGBITS          0000000000000000 00000058
     0000000000000024 MS 0 0 1
 [ 5] .note.GNU-stack    PROGBITS          0000000000000000 00000058
     0000000000000000 0 0 1
 [ 6] .eh_frame           PROGBITS          0000000000000000 00000088
     0000000000000040 A 0 0 8
 [ 7] .rela.eh_frame      RELA             0000000000000000 000004f8
     0000000000000030 9 6 8
 [ 8] .shstrtab           STRTAB           0000000000000000 000000c8
     0000000000000054 0 0 1
 [ 9] .symtab             SYMTAB           0000000000000000 000003e0
     00000000000000f0 10 8 8
 [10] .strtab             STRTAB           0000000000000000 000004d0
     0000000000000027 0 0 1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
[sharexu@linux elf_example]$
```

图 4-19 使用 readelf 工具来查看可重定位对象文件 add.o 的 section 表内容

注意上面结果中的 Flags，表示的是对应 section 的相关标志。比如 .text section 里面存储的是代码，所以就是可执行的（用 X 表示）；.data 和 .bss 里面存放的都是可写的（用 W 表示）数据（非在堆栈中定义的数据），只不过前者存的是初始化过的数据，如程序中定义的赋过初值的全局变量等；而后者里面存储的是未经过初始化的数据。因为未经过初始化就意味着不确定这些数据刚开始的时候会有些什么样的值，所以针对对象文件来说，它就没必要为了存储这些数据而在文件内多留出一块空间，因此 .bss section 的大小总是为 0。后面会看到，当可执行程序被执行的时候，动态连接器会在内存中开辟一定大小的空间来存放这些未初始化的数据，里面的内存单元都被初始化成 0。可执行程序文件中虽然没有长度非 0 的 .bss section，但却记录着在程序运行时需要开辟多大的空间来容纳这些未初始化的数据。

另外一个标志 A 说明对应的 section 是 Allocable（可分配的）的。所谓可分配的 section，是指在运行时，进程（process）需要使用它们，所以它们被加载器加载到内存中去。

而与此相反，存在一些 non-Allocable 的 sections，它们只是被链接器、调试器或者其他类似工具所使用的，而并非参与进程的运行中去。如后面要介绍的字符串表 section .strtab，符号表 .symtab section 等。当运行最后的可执行程序时，加载器会加载那些 Allocable 的部分，而 non-Allocable 的部分则会被继续留在可执行文件内。所以，实际上，这些 non-Allocable 的 section 都可以被 strip 工具从最后的可执行文件中删除掉，删除掉这些 sections 的可执行文件依然能够运行，只不过没办法来进行调试之类的操作罢了。

3. ELF 的 .text section

可以使用 readelf -x SecNum 来打印出不同 section 中的内容。但是，无奈其输出结果都是机器码，对人来说不具备可读性。所以可以换用另外一个工具 objdump 来看看这些 sections 中到底具有哪些内容。执行命令：objdump -d -j .text add.o 得到如图 4-20 所示的

结果。

```
[sharexu@linux elf_example]$ objdump -d -j .text add.o
add.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <_Z3addii>:
0: 55                push    %rbp
1: 48 89 e5          mov     %rsp,%rbp
4: 89 7d fc          mov     %edi,-0x4(%rbp)
7: 89 75 f8          mov     %esi,-0x8(%rbp)
a: 8b 45 f8          mov     -0x8(%rbp),%eax
d: 8b 55 fc          mov     -0x4(%rbp),%edx
10: 8d 04 02          lea     (%rdx,%rax,1),%eax
13: c9               leaveq  (%rdx,%rax,1),%eax
14: c3               retq
```

图 4-20 命令 `objdump -d -j .text add.o` 的执行结果

`objdump` 的选项 `-d` 表示要对由 `-j` 选择项指定的 section 内容进行反汇编，也就是由机器码出发，推导出相应的汇编指令。上面结果显示在 `add.o` 目标文件的 `.text` 中只是包含了函数 `add`（编译后，函数名变为 `_Z3addii`）的定义。

4. ELF 的 .data section

用同样的方法，下面来看看 `add.o` 中 `.data` section 有什么内容，执行 `objdump -d -j .data add.o` 命令，可得到如图 4-21 所示的结果。

这个结果显示，在 `add.o` 的 `.data` section 中并没有定义任何变量。假如把 `add.cpp` 中加入一个变量，并初始化为 12，即 `add.cpp` 的内容变成这样：

```
#include "add.h"
int result=12;
int add(int a,int b){
    return a+b;
}
```

重新生成 `add.o` 文件后，执行 `objdump -d -j .data add.o` 命令，可得到如图 4-22 所示的结果。

```
[sharexu@linux elf_example]$ objdump -d -j .data add.o
add.o:      file format elf64-x86-64
```

图 4-21 `objdump -d -j .data add.o` 的执行结果

```
[sharexu@linux elf_example]$ objdump -d -j .data add.o
add.o:      file format elf64-x86-64

Disassembly of section .data:

0000000000000000 <result>:
0: 0c 00 00 00      ....
```

图 4-22 `objdump -d -j .data add.o` 的执行结果

这个结果显示在 `add.o` 的 `.data` section 中定义了一个 4 Byte 的变量，其值被初始化成 `0x0000000c`，也就是十进制值 12。之前 `file add.o` 时可以看到本书的测试机是 x86 架构的，

而 x86 架构是使用小端模式，所以存储为 0c000000。再来看下修改后的 add.o 的 section 表都变成什么样子。

从图 4-23 可以看出，原先 .bss section 的 offset 位置是在 00000058，和 .data section 的 offset 位置一样，这意味着，以前 .data section 数据为空。而现在，.bss 的 offset 位置是在 0000005c，.data section 的 offset 位置是在 00000058，则 0000005c-00000058=0x4，也就是 4 Byte，这正好是 result 这个 int 变量刚好需要的 4 Byte，所以 result 变量就存在 .data section 中。（readelf 命令使用 -S 参数可以列出 elf 文件的所有节的信息。）

```
[sharexu@linux elf_example]$ readelf -S add.o
There are 11 section headers, starting at offset 0x128:

Section Headers:
 [Nr] Name              Type              Address            Offset
     Size              EntSize          Flags Link Info  Align
 [ 0] 0000000000000000 NULL              0000000000000000 0 0 0
 [ 1] 0000000000000015 .text             PROGBITS          0000000000000000 0 0 0 4
 [ 2] 0000000000000015 .data             PROGBITS          0000000000000000 0 0 0 4
 [ 3] 0000000000000004 .bss              NOBITS            0000000000000000 0 0 0 4
 [ 4] 0000000000000000 .comment           PROGBITS          0000000000000000 0 0 0 1
 [ 5] 000000000000002d .note.GNU-stack   PROGBITS          0000000000000000 0 0 0 1
 [ 6] 0000000000000000 .eh_frame          PROGBITS          0000000000000000 0 0 0 8
 [ 7] 0000000000000040 .rela.eh_frame     RELA              0000000000000000 9 6 8
 [ 8] 0000000000000030 .shstrtab          STRTAB            0000000000000000 0 0 1
 [ 9] 0000000000000054 .symtab            SYMTAB            0000000000000000 10 8 8
[10] 0000000000000108 .strtab            STRTAB            0000000000000000 0 0 1
     000000000000002e 0000000000000000

Key to Flags:
 W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), x (unknown)
 0 (extra OS processing required) o (OS specific), p (processor specific)
[sharexu@linux elf_example]$
```

图 4-23 修改后的 add.o 的 section 表

5. ELF 的 .strtab section

接下来将介绍字符串表 .strtab section 的相关内容。执行 readelf -x 10 add.o 结果如图 4-24 所示。

readelf -x num file 以 16 进制方式显示指定段内内容。num 指定段表中段的索引，或字符串指定文件中的段名。这里 num 为 10，是因为 readelf -S add.o 后看到 .strtab section 的索引是 10。

上面结果中的十六进制数据部分从右到左看是地址递增的方向，而字符内容部分从左到右看是地址递增的方向。所以，在 .strtab section 中，按照地址递增的方向来看，各字节的内容依次是 0x00、0x61、0x64、0x64、0x2e、0x63、0x70、0x70...，也就是字符“、‘a’、‘d’、‘d’、‘!’、‘c’、‘p’、‘p’... 等。也可以使用 hexdump 直接 dumping 出 .strtab section 开头（其偏移在文件内 0x4f0 字节处）的 32 Byte 数据，输入以下命令 hexdump -s 0x508 -n 32 -c add.o 得到，如图 4-25 所示的结果。


```
[sharexu@linux elf_example]$ readelf -x 10 add.o
```

```
Hex dump of section '.strtab':
0x00000000 00616464 2e637070 00726573 756c7400 .add.cpp.result.
0x00000010 5f5a3361 64646569 005f5f67 78785f70 _Z3addii__gxx_p
0x00000020 6572736f 6e616c69 74795f76 3000     _personality_v0.
[sharexu@linux elf_example]$
```

图 4-24 命令 `readelf -x 10 add.o` 的执行结果

```
[sharexu@linux elf_example]$ hexdump -s 0x508 -n 32 -c add.o
```

```
0000508  \0  -  g  x  x  p  e  r  s  o  n  a  l  i
0000518  t  y  -  v  0  \0  \0  \0  022  \0  \0  \0  \0  \0  \0
0000528
[sharexu@linux elf_example]$
```

图 4-25 命令 `hexdump -s 0x508 -n 32 -c add.o` 的执行结果

这里的 0x508 是 `readelf -S add.o` 后看到 `.strtab` section 的 offset 值。

`.strtab` section 中存储着的都是以字符为分隔符的字符串，这些字符串所表示的内容，通常是程序中定义的函数名称、所定义过的变量名称等。当对象文件中其他地方需要和一个这样的字符串相关联的时候，往往会在对应的地方先存储 `.strtab` section 中的索引值。比方下面将要介绍的符号表 `.symtab` section 中，有一个条目是用来描述符号 `result` 的，那么在该条目中就会有一个字段 (`st_name`) 用来记录着字符串 `result` 在 `.strtab` section 中的索引 7。`.shstrtab` 也是字符串表，只不过其中存储的是 section 的名字，而非函数或者变量的名称。

6. ELF 的 `.symtab` section

字符串表在真正链接和生成进程映像过程中是不需要使用的，但是其对我们调试程序来说就特别有帮助，因为从人的角度来看起来最舒服的还是自然形式的字符串，而非天书一样的数字符号。前面使用 `objdump` 来反汇编 `.text` section 的时候，之所以能看到定义了函数 `add`，那也是因为这个字符串表的原因。当然起关键作用的，还是符号表 `.symtab` section 在其中作为中介，下面就来看看符号表。

虽然同样可以使用 `readelf -x` 来查看符号表 (`.symtab`) section 的内容，但是其结果可读性太差，所以换用 `readelf -s` 或者 `objdump -t` 来查看。执行 `readelf -s add.o` 命令得到结果，如图 4-26 所示。

```
[sharexu@linux elf_example]$ readelf -s add.o
Symbol table '.symtab' contains 11 entries:
Num:  Value              Size Type Bind Vis Ndx Name
0: 0000000000000000      0 NOTYPE LOCAL DEFAULT UND
1: 0000000000000000      0 FILE LOCAL DEFAULT ABS add.cpp
2: 0000000000000000      0 SECTION LOCAL DEFAULT 1
3: 0000000000000000      0 SECTION LOCAL DEFAULT 2
4: 0000000000000000      0 SECTION LOCAL DEFAULT 3
5: 0000000000000000      0 SECTION LOCAL DEFAULT 5
6: 0000000000000000      0 SECTION LOCAL DEFAULT 6
7: 0000000000000000      0 SECTION LOCAL DEFAULT 4
8: 0000000000000000      4 OBJECT GLOBAL DEFAULT 2 result
9: 0000000000000000     21 FUNC GLOBAL DEFAULT 1 _Z3addii
10: 0000000000000000      0 NOTYPE GLOBAL DEFAULT UND __gxx_personality_v0
[sharexu@linux elf_example]$
```

图 4-26 命令 `readelf -s add.o` 的执行结果

`readelf -s file`，显示符号表段中的项。

在符号表内针对每一个符号，都会相应的设置一个条目。比如变量 `result` 的类型就是 `OBJECT`；而函数 `add`（编译后得到的函数名为 `_Z3addii`）的类型是 `FUNC`。

4.3.3 执行视图下的 ELF 内容

接下来将介绍可执行文件的程序头部。执行 `readelf -l main` 命令后，得到的结果如

图 4-27 所示。

```
[sharexu@linux_elf_example]$ readelf -l main
Elf file type is EXEC (Executable file)
Entry point 0x400800
There are 8 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
-----
FileSiz      MemSiz      Flags  Align
PHDR          0x0000000000000040 0x0000000000000040 0x0000000000000040
0x00000000000001c0 0x00000000000001c0 R E    8
INTERP        0x0000000000000200 0x0000000000000200 0x0000000000000200
0x000000000000021c 0x000000000000021c R      1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD          0x0000000000000000 0x0000000000000000 0x0000000000000000
0x00000000000000ac 0x00000000000000ac R E    200000
LOAD          0x0000000000000100 0x0000000000000100 0x0000000000000100
0x000000000000027c 0x000000000000027c R W    200000
DYNAMIC       0x0000000000000100 0x0000000000000100 0x0000000000000100
0x00000000000001d0 0x00000000000001d0 R W    8
NOTE          0x000000000000021c 0x000000000000021c 0x000000000000021c
0x0000000000000044 0x0000000000000044 R      4
GNU_EH_FRAME  0x0000000000000b94 0x0000000000000b94 0x0000000000000b94
0x0000000000000034 0x0000000000000034 R      4
GNU_STACK     0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000000000 0x0000000000000000 R W    8

Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .interp.note.ABI-tag.note.gnu.build-id.gnu.hash.dynsym.dynstr.gnu.version.gnu.version_r
.rela.dyn.rela.plt.init.plt.text.fini.rodatab.eh_frame_hdr.eh_frame
03 .ctors.dtors.jcr.dynamic.got.got.plt.data.bss
04 .dynamic
05 .note.ABI-tag.note.gnu.build-id
06 .eh_frame_hdr
07
```

图 4-27 命令 `readelf -l main` 的执行结果

`readelf -l main`，显示 `main` 程序头的信息。

结果显示，在可执行文件 `main` 中，总共有 8 个 segments (program headers)。同时，该结果也很明白显示出了哪些 section 映射到哪一个 segment 当中去。比方在索引为 2 的那个 segment 中，总共有 16 个 sections 映射进来，其中包括在前面提到过的 `.text` section。注意这个 segment 有两个标志：R 和 E。这个表示该 segment 是可读的、可执行的。如果看到标志中有 W，那表示该 segment 是可写的。

类型为 `INTERP` 的 segment 只包含一个 section，那就是 `.interp`。在这个 section 中，包含了动态链接过程中所使用的解释器路径和名称。在 Linux 里面，这个解释器实际上就是 `/lib/`，执行命令 `hexdump -s 0x200 -n 32 -C main` 可以得到如图 4-28 所示的结果。

```
[sharexu@linux_elf_example]$ hexdump -s 0x200 -n 32 -C main
00000200 2f 6c 69 62 36 34 2f 6c 64 2d 6c 69 6e 75 78 2d |/lib64/ld-linux-|
00000210 78 38 36 2d 36 34 2e 73 6f 2e 32 00 04 00 00 00 |x86-64.so.2.....|
00000220
[sharexu@linux_elf_example]$
```

图 4-28 命令 `hexdump -s 0x200 -n 32 -C main` 的执行结果

这里的 `0x200` 是 `INTERP` 的 Offset 值。

为什么会有这样的一个 segment？这是因为我们写的应用程序通常都需要使用动态链接库 `.so`，就像 `test` 程序中所使用的 `libsub.so` 一样。程序在 Linux 里面运行时，当在 shell 中键入一个命令要执行时，内核会创建一个新的进程，在往这个新进程的进程空间里面加载进可执

行程序的代码段和数据段后，也会加载进动态连接器（在 Linux 里面通常就是 `/lib/ld-linux.so` 符号链接所指向的那个程序，它本身就是一个动态库）的代码段和数据。在这之后，内核将控制传递给动态链接库里面的代码。动态连接器接下来负责加载该命令应用程序所需要使用的各种动态库。加载完毕，动态连接器才将控制传递给应用程序的 `main` 函数。如此操作后相应的应用程序才得以运行。

关于其他 `segment`，本章先不展开，等后面讲关于进程一章时再进行探索。

4.3.4 阅读 ELF 文件的工具——`readelf`

如前面所示，`readelf` 命令用来显示一个或者多个 `elf` 格式的目标文件的信息，可以通过它的选项来控制显示哪些信息。本节再简单地总结下它的用法，如下所示。

```
readelf -v           // 显示版本
readelf -h           // 显示帮助
readelf -a test      // 显示 test 的全部信息
readelf -h test      // 显示 test 的 ELF Header 的文件头信息（就是 ELF 文件开始的前 52 Byte）
readelf -l test      // 显示 test 的 Program Header Table 中的每个 Program Header Entry 的信息（如果有）
readelf -S test      // 显示 test 的 Section Header Table 中的每个 Section Header Entry 的信息（如果有）
readelf -g test      // 显示 test 的 Section Group 的信息（如果有）
readelf -s test      // 显示 test 的 Symbol Table 中的每个 Symbol Table Entry 的信息（如果有）
readelf -e test      // 显示 test 的全部头信息（包括 ELF Header, Section Header 和 Program Header, 等同与 readelf -h -l -S test）
readelf -n test      // 显示 test 的 note 段的信息（如果有）
readelf -r test      // 显示 test 中的可重定位段的信息（如果有）
readelf -d test      // 显示 test 中的 Dynamic Section 的信息（如果有）
readelf -V test      // 显示 test 中的 GNU Version 段信息（如果有）
```

`readelf` 和 `objdump` 提供的功能类似，但是它显示的信息更为具体，并且它不依赖 BFD 库（BFD 库是一个 GNU 项目，它的目标就是希望通过一种统一的接口来处理不同的目标文件），所以即使 BFD 库有什么 bug 存在的话也不会影响到 `readelf` 程序。运行 `readelf` 的时候，除了 `-v` 和 `-H` 之外，其他的选项必须至少有一个被指定。

4.3.5 获得二进制文件里符号的工具——`nm`

`nm` 是用来查看指定程序中的符号表相关内容的工具。通过例 4.3 来介绍 `nm` 工具，先看一下这个简单的程序。

【例 4.3】熟悉 `nm` 使用的例子。

`test.cpp` 的代码是：

```
#include<iostream>
using namespace std;
```

```

class Test{
public:
    int Hello(){
        cout<<"Hello world!"<<endl;
        return 0;
    }
};

int main(){
    Test test;
    int iRet=test.Hello();
    cout<<"iRet="<<iRet<<endl;
    return 0;
}

```

makefile 的代码是:

```
test:test.o
    g++ test.o -o test
test.o:test.cpp
    g++ -c test.cpp
```

编译该程序，然后看 nm 的结果，如图 4-29 所示。

```
0000000000000000 example$ nm test
0000000000000008 b DYNAMIC
0000000000000060 d GLOBAL_OFFSET_TABLE_
0000000000004008 b T GLOBAL_I_main
00000000000040a0 R IO_stdin_used
0000000000004008 w JV_RegisterClasses
0000000000004008 b T Zdl_static_initialization_and_destruction_0ii
000000000000408f0 W ZN4TestShellToEv
U ZNSolsEFPFRSoS_F@GLIBCXX_3.4
U ZNSolsEia@GLIBCXX_3.4
U ZNST8ios_baseInitCIEv@GLIBCXX_3.4
U ZNST8ios_baseInitDIEv@GLIBCXX_3.4
00000000000060d0 B ZSt4cout@@GLIBCXX_3.4
U ZSt4endlc1lchar_traitsIcEERSt13basic_ostreamIT_T_E56_@@GLIBCXX_3.4
0000000000006f00 b ZStL8_oinit
U ZSt15Ittllchar_traitsIcEERSt13basic_ostreamIC_T_E55_Pkc@@GLIBCXX_3.4
0000000000006078 d _CTOR_END_
0000000000006068 d _CTOR_LIST_
0000000000006088 d DTOR_END_
0000000000006080 d DTOR_LIST_
0000000000004060 r FRAME_END_
0000000000006090 d JCR_END_
0000000000006090 d JCR_LIST_
00000000000060dc4 A __bss_start
U __cxa_atexit@@GLIBC_2.2.5
00000000000060dc0 d __data_start
0000000000004090 c t do_global_ctors_aux
00000000000040070 c t do_global_dtors_aux
00000000000040a10 R __dso_handle
w __gmon_start__
U __gxx_personality_v0@@GCCABI_1.3
00000000000060b4 d __init_array_end
00000000000060b4 d __init_array_start
00000000000040920 T __libc_csu_fini
00000000000040930 T __libc_csu_init
U __libc_start_main@@GLIBC_2.2.5
00000000000060dc4 A _edata
00000000000050f08 A _end
000000000000409f8 T _fini
000000000000406b8 T __init
00000000000040770 T __start
0000000000004079c t call_gmon_start
00000000000060ef0 b completed.6347
00000000000060dc0 W __data_start
00000000000060ef8 b dtor_idx.6349
00000000000040830 t frame_dummy
00000000000040854 T main
[sharexug@linux nm example$ ]
```

图 4-29 执行 nm 命令的结果图

上面便是 `test` 这个程序中所有的符号，首先需要介绍一下上面的内容的格式：①第一列是当前符号的地址；②第二列是当前符号的类型；③第三列是当前符号的名称。

在上面的结果中，像 `_ZN4Test5HelloEv` 这样的符号，很多读者朋友可能会不明白其含义，这里介绍个小技巧，在执行 `nm` 命令的时候，加上 `-C` 选项，就可以把这些难以识别的符号，转换成便于阅读的符号 `TestHello()`。这个主要是 C++ 中的 `mangle` 机制所导致的，加上 `-C` 就是指定列出的符号是 `demangle` 了的。说了这么多，其实 `nm` 命令对程序的帮助，主要有以下几个方面。

1) 判断指定程序中有没有定义指定的符号（比较常用的方式：`nm -C proc | grep symbol`）。

2) 解决程序编译时 `undefined reference` 的错误，以及 `mutiple definition` 的错误。

3) 查看某个符号的地址，以及在进程空间的大概位置（`bss`、`data`、`text` 区，具体可以通过第二列的类型来判断）。

4.3.6 减少目标文件大小的工具——strip

UNIX 下文件压缩命令有 `compress` 和 `tar`，结合使用来做数据备份是最合适不过了。但 `compress` 压缩也有缺点，就是被压缩后的文件需要用命令 `uncompress` 解压后才能正常使用；而用 `strip` 命令就没有这个问题，它能清除执行文件中不必要的标示符及调试信息，可减小文件大小而不影响正常使用。但与 `compress` 不同的是，文件一旦进行 `strip` 操作后就不能恢复原样了，所以 `strip` 可以认为是一个“减肥”工具而不是压缩工具。而且，被 `strip` 后的文件不包含调试信息。现在来看具体效果如何，还用例 4.3 中生成的目标文件 `test`，对其进行 `strip` 操作后得到 4-30 所示的结果。

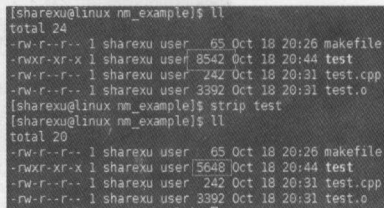
可见，执行 `strip test` 命令后，`test` 的文件大小由 8542bit 减少到 5648bit，减少了近 1/3。`strip` 命令能从 ELF 文件中有选择地除去行号信息、重定位信息、调试段、`typchk` 段、注释段、文件头以及所有或部分符号表。但一旦使用该命令，则很难调试文件的符号；因此，通常只在已经调试和测试过的生成模块上使用 `strip` 命令，来减少对象文件所需的存储量开销。

其他常用选项如下所述。

(1) `-l` (小写 L)：从对象文件中除去行号信息。

(2) `-r`：除了外部符号和静态符号条目，将全部符号表信息除去。不除去重定位信息。同时除去调试段和 `typchk` 段。这个选项产生一个对象文件，该对象文件仍可以用作输入到链接编辑器中。

(3) `-t`：除去大多数符号表信息，但并不除去函数符号或行号信息。



```
[sharexu@linux nm_example]$ ll
total 24
-rw-r--r-- 1 sharexu user 8542 Oct 18 20:26 makefile
-rwxr-xr-x 1 sharexu user 8542 Oct 18 20:44 test
-rw-r--r-- 1 sharexu user 242 Oct 18 20:31 test.cpp
-rw-r--r-- 1 sharexu user 3392 Oct 18 20:31 test.o
[sharexu@linux nm_example]$ strip test
[sharexu@linux nm_example]$ ll
total 20
-rw-r--r-- 1 sharexu user 65 Oct 18 20:26 makefile
-rwxr-xr-x 1 sharexu user 5648 Oct 18 20:44 test
-rw-r--r-- 1 sharexu user 242 Oct 18 20:31 test.cpp
-rw-r--r-- 1 sharexu user 3392 Oct 18 20:31 test.o
```

图 4-30 执行 `strip` 命令后可执行文件变小了

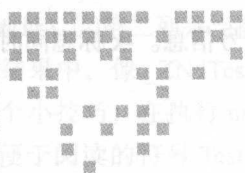
(4) -V: 打印 strip 命令的版本号。

(5) -x: 除去符号表信息,但并不除去静态或外部符号信息。-x 标志同时除去重定位信息,因此将不可能链接到该文件。

4.4 本章小结

本章讲述了编译与链接、自动化编译利器 makefile 的使用、目标文件及其相关工具等相关知识,掌握这些知识后,你就已经具备了完成大型功能开发的能力了。

程序编译成功后,还需要进行调试,第5章将开始学习如何调试。



调 试

在理想世界里，每当一个程序不能正常执行某个功能时，它就会给出一个有用的错误提示，告诉开发者足够的改正错误的线索。但遗憾的是，在现实世界中，很多时候一个程序出现了问题时却无法找到原因。这就是调试程序出现的原因。

调试的方法一般有 2 种，如下所述。

(1) 在程序中插入打印语句，优点是能够显示程序的动态过程，比较容易检查源程序的有关信息。缺点是效率低，可能输入大量无关的数据，发现错误具有偶然性。

(2) 借助调试工具。目前大多数程序设计语言都有专门的调试工具，比如 C++ 的调试工具有 GDB，可以用这些工具来分析程序的动态行为。

本章主要讲一些调试工具，方便读者更高效率地进行调试。

5.1 strace

1. 系统调用

为了为创建文件、进程和复制文件等这些操作系统提供的服务，应用程序必须和操作系统之间进行交互。但是，应用程序是不能直接访问 Linux 内核的。它既不能访问内核所占内存空间，也不能调用内核函数。不过，应用程序可以跳转到 `system_call` 的内核位置，内核会检查系统调用号，这个号码会告诉内核进程正在请求哪种服务。然后，它查看系统调用表，找到所调用的内核函数入口地址，调用该函数，然后返回到进程。

所有操作系统在其内核都有一些内建的函数，这些函数可以用来完成一些系统级别的功能，一般称 Linux 系统上的这些函数为“系统调用”（system call）。这些函数代表了用户

空间到内核空间的一种转换,例如,在用户空间调用 `open` 函数,在内核空间则会调用 `sys_open`。

系统调用的错误码:系统调用并不直接返回错误码,而是将错误码放入一个名为 `errno` 的全局变量中。如果一个系统调用失败,你可以读出 `errno` 的值来确定问题的所在。`errno` 不同数值所代表的错误消息定义在 `errno.h` 中,你也可以通过命令 "`man 3 errno`" 来查看它们。

需要注意的是, `errno` 的值只在函数发生错误时设置,如果函数不发生错误, `errno` 的值就无定义,并不会被置为 0。另外,在处理 `errno` 前最好先把它的值存入另一个变量,因为在错误处理过程中,即使像 `printf()` 这样的函数出错时也会改变 `errno` 的值。

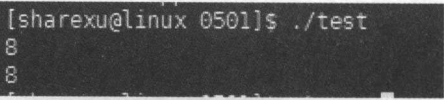
而 `strace` 就是一个通过跟踪系统调用来让开发者知道一个程序在后台所做事情的工具。

2. strace 初识

先来用一个简单的程序来演示 `strace` 的基本用法,如例 5.1 所示。

【例 5.1】 输入一个数,并输出这个数。

```
#include <iostream>
using namespace std;
int main(){
    int a;
    cin>>a;
    cout<<a<<endl;
    return 0;
}
```



```
[sharexu@linux 0501]$ ./test
8
8
```

然后用 `g++ -o test test.cpp` 编译一下,得到一个可执行文件 `test`,执行结果如图 5-1 所示。

图 5-1 执行 `./test` 命令得到的结果图

输入了一个数 8,也输出了一个数 8。然后用 `strace` 调用执行,得到的结果如图 5-2 和图 5-3 所示。

每一行都是一次系统调用,等号左边是系统调用的函数名及其参数,右边是该调用的返回值。从 `strace` 结果可以看到,系统首先调用 `execve`,以开始一个新的进行,接着进行一些环境的初始化操作,最后停顿在 `read(0)` 上面,这也就是执行到了 `cin` 函数后,等待用户输入数字。在输入完“8”之后,再调用 `write` 函数将格式化后的数值 8 输出到屏幕,最后调用 `exit_group` 退出进行,完成整个程序的执行过程。

接下来再选择部分结果来做详细分析:

```
execve("./test", [ "./test" ], [ /* 22 vars */ ]) = 0
```

对于命令行下执行的程序, `execve` (或 `exec` 系列调用中的某一个) 均为 `strace` 输出系统调用中的第一个。`strace` 首先调用 `fork` 或 `clone` 函数新建一个子进程,然后在子进程中调用 `exec` 载入需要执行的程序 (这里为 `./test`)。


```
brk(0)
```

以 0 作为参数调用 `brk`，返回值为内存管理的起始地址（若在子进程中调用 `malloc`，则从 `0x1787000` 地址开始分配空间）。

```
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f97c2899000
```

使用 `mmap` 函数进行匿名内存映射，以此来获取 4096Bytes 内存空间，该空间起始地址为 `0x7f97c2899000`，匿名内存映射就是为了不涉及具体的文件名，避免了文件的创建及打开，这种只能用于具有亲缘关系的进程间通信。关于进程间通信后面的章节会有详细描述，

这里暂不展开。

```
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
```

调用 `access` 函数检验 `/etc/ld.so.preload` 是否存在。

```
open("/etc/ld.so.cache", O_RDONLY) = 3
```

调用 `open` 函数尝试打开 `/etc/ld.so.cache` 文件，返回文件描述符为 3。

```
fstat(3, {st_mode=S_IFREG|0644, st_size=21000, ...}) = 0
```

使用 `fstat` 函数获取 `/etc/ld.so.cache` 文件信息。

```
mmap(NULL, 21000, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f97c2893000
```

调用 `mmap` 函数将 `/etc/ld.so.cache` 文件映射至内存。

```
close(3)
```

`close` 关闭文件描述符为 3 指向的 `/etc/ld.so.cache` 文件。

```
open("/usr/lib64/libstdc++.so.6", O_RDONLY) = 3
```

```
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360c\5\0\0\0\0"..., 832) = 832
```

调用 `open` 和 `read`，从 `/usr/lib64/libstdc++.so.6` 该 `libc` 库文件中读取 832Bytes，即读取 ELF 头信息。上一章讲到目标文件的 ELF 头部中有进程的进入点，这里就是在获得进程的进入点。

```
mprotect(0x7f97c245d000, 2097152, PROT_NONE) = 0
```

使用 `mprotect` 函数对 `0x7f97c245d000`，起始的 2097152Bytes 空间进行保护（`PROT_NONE` 参数就是不能访问，对应还有 `PROT_READ` 表示可以读取）。

```
munmap(0x7f97c2893000, 21000) = 0
```

调用 `munmap` 函数，将 `/etc/ld.so.cache` 文件从内存中去映射，与下面这行的 `mmap` 对应。

```
mmap(NULL, 21000, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f97c2893000
```

对应源码中使用到的两个系统调用——`read` 函数，读取从终端输入的内容和输出内容到终端：

```
read(0, 8
```

```
"8\n", 1024)
```

```
= 2
```

```
write(1, "8\n", 28
```

```
)
```

```
= 2
```

子进程结束，退出码为 0。

```
exit_group(0)
```

从源码看来，真正能与源码对应上的只有 `read` 和 `write` 这两个系统调用，其他系统调用几乎都用于进行进程初始化工作：装载被执行程序、载入 `libc` 函数库、设置内存映射等。

再看看可能 `strace` 到的常用的系统调用。

有错误产生时，一般会返回 `-1`，所以会有错误标志和描述，例如：

```
open(\\."/for/bar\\",)_RDONLY) = -1 ENOENT (no such file or directory)
```

这个表示打开 `/for/bar` 文件错误，后面的描述提示是没有这个文件。

`char*` 将作为 C 的字符串类型输出。没有字符串输出时一般是 `char*`，是一个转义字符，只输出字符串的长度。当字符串过长时会使用 `\\` 部分字符串 `\\` 省略（如在 `\\` " `ls -\\` " 会有一个 `gepwuid` 调用读取 `password` 文件）：

```
read(3, \\ "root::0:0:System Administrator:\\ " ...,1024) = 422
```

当参数是结构数组时，将按照简单的指针和数组输出，代码如下：

```
getgroups(4,[0,2,4,5]) = 4
```

关于 `bit` 作为参数的情形，也是使用方括号，并且用空格将每一项参数隔开，代码如下：

```
sigprocmask(SIG_BLOCK,[CHLD TTOU],[ ]) = 0
```

这里第二个参数代表两个信号 `SIGCHLD` 和 `SIGTTOU`。如果 `bit` 型参数全部置位，则有如下的输出，代码如下：

```
sigprocmask(SIG_UNBLOCK,~[ ],NULL) = 0
```

这里第二个参数全部置位。

3. 用 `strace` 来跟踪信号传递

还可以用 `strace` 来跟踪信号传递。这里还是使用例 5.1 的那个 `test` 程序，来观察进程接收信号的情况。还是先输入命令 “`strace ./test`”，等待的时候不要输入任何东西，然后打开另外一个窗口，输入命令 “`killall test`”，执行结果如图 5-4 和图 5-5 所示。

`strace` 中的结果显示 `test` 进程 “+++ killed by SIGTERM +++”。事实上，命令 `killall test`，就是杀死所有名为 `test` 的进程。

4. 统计系统调用

`strace` 不光能追踪系统调用，通过使用参数 `-c`，它还能将进程所有的系统调用做一个统计分析并返回。下面就来看看 `strace` 的统计，这次执行带 `-c` 参数的 `strace`，执行 `strace -c test` 命令后，能得到如图 5-6 所示的结果。

如果例 5.1 中的 `cin` 和 `cout` 都换成了 C 中的 `scanf` 和 `printf`，那么其系统调用情况是否会一样，下面就来验证一下。

【例 5.2】 用 `scanf` 和 `printf` 输入一个数，并输出这个数。

```
#include<stdio.h>
int main(){
    int a;
    scanf("%d",&a);
    printf("%d\n",a);
    return 0;
}
```

还是用 `g++ -o test test.cpp` 对其进行编译，用 `strace -c` 来统计其系统调用次数，结果如图 5-7 所示。

如图 5-7 中所示，用 C++ 的 `cin` 输入 `cout` 输出和 C 的 `scanf` 输入 `printf` 输出，用到的系统调用数量都是一样的。

```
[sharexu@linux 0502]$ strace -c ./test
0
0
% time      seconds  usecs/call   calls   errors syscall
-----
-nan 0.000000      0          5        read
-nan 0.000000      0          1        write
-nan 0.000000      0          5        open
-nan 0.000000      0          5        close
-nan 0.000000      0          7        fstat
-nan 0.000000      0         18        mmap
-nan 0.000000      0          8        mprotect
-nan 0.000000      0          1        munmap
-nan 0.000000      0          1        brk
-nan 0.000000      0          1        l access
-nan 0.000000      0          1        execve
-nan 0.000000      0          1        arch_prctl
-----
100.00 0.000000          54        1 total
[sharexu@linux 0502]$
```

图 5-7 用 `strace` 统计 `scanf` 输入 `printf` 输出程序发生的系统调用结果图

5. 其他常用选项

除了 `-c` 参数之外，`strace` 还提供了其他有用的参数，能很方便地得到自己想要的信息，下面就对那些常用的参数一一介绍。

参数 `-o` 用在将 `strace` 的结果输出到文件中，如果不指定 `-o` 参数的话，默认的输出设备是 `STDERR`，也就是说使用“`-o filename`”和“`2>filename`”语句的结果是一样的，如图 5-8 所示。

把 `-o` 指定的输出文件 `test.txt` 和 `2>test2.txt` 的结果做比较（用 `diff` 命令），可以看到两个文件一样，也就是说这两个命令是等价的。

`strace` 可以使用参数 `-T` 将每个系统调用所花费的时间打印出来，每个调用的时间花销都体现在调用行最右

```
[sharexu@linux 0501]$ strace -c -o test.txt ./test
8
8
[sharexu@linux 0501]$ strace -c ./test 2>test2.txt
8
8
[sharexu@linux 0501]$ diff test.txt test2.txt
[sharexu@linux 0501]$
```

图 5-8 比较 `strace` 使用“`-o filename`”和“`2>filename`”的结果

边的尖括号里面。执行 `strace -T ./test` 可以得到如图 5-9 所示（这里只截取了前面几行）。

```
[sharexu@linux 0501]$ strace -T ./test
execve("./test", ["/test"], [/* 22 vars */]) = 0 <0.000096>
brk(0) = 0x257c000 <0.000008>
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f31d3e98000 <0.000010>
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory) <0.000010>
open("/etc/ld.so.cache", O_RDONLY) = 3 <0.000013>
fstat(3, {st_mode=S_IFREG|0644, st_size=21000, ...}) = 0 <0.000008>
mmap(NULL, 21000, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f31d3e92000 <0.000010>
close(3) = 0 <0.000008>
open("/usr/lib64/libstdc++.so.6", O_RDONLY) = 3 <0.000013>
```

图 5-9 执行 `strace -T` 命令可得到每个系统调用所花费的时间

结果表明，调用 `execve` 函数花了 0.000096s，调用 `mmap` 函数花了 0.000010s。

`strace` 的 `-t`、`-tt`、`-ttt` 参数则是记录每次系统调用发生的时间，分别精确到秒、微秒和 UNIX 时间戳的微秒。执行 `strace -t ./test` 命令可以得到如图 5-10 所示的结果（这里只截取了前面几行）。

```
[sharexu@linux 0501]$ strace -t ./test
22:34:39 execve("./test", ["/test"], [/* 22 vars */]) = 0
22:34:39 brk(0) = 0x1544000
22:34:39 mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9e4537e000
22:34:39 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
22:34:39 open("/etc/ld.so.cache", O_RDONLY) = 3
22:34:39 fstat(3, {st_mode=S_IFREG|0644, st_size=21000, ...}) = 0
22:34:39 mmap(NULL, 21000, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f9e45378000
22:34:39 close(3) = 0
```

图 5-10 执行 `strace -t` 命令可记录每次系统调用发生的秒级别时间

由图 5-10 可知，这几个系统调用都是在 22:34:39 这一秒内发生的。

`strace` 不光能自己初始化一个进程进行 `strace`，还能追踪现有的进程，参数 `-p` 就是取这个作用的，用法也很简单，具体如下：

```
strace -p pid
```

其中，`pid` 是指进程 id。

6. 用 `strace` 调试程序

接下来举一个例子看看怎么用 `strace` 来调试程序。

【例 5.3】 用 `strace` 调试程序。

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
using namespace std;
int main(){
    char buffer[256];
    ifstream in("input.txt");
    if (!in.is_open()){
        cout << "Error opening file"<<endl;
        exit (1);
    }
    while (!in.eof()){
        in.getline (buffer,100);
```



```

    cout << buffer << endl;
}
return 0;
}

```

例 5.3 中的程序是从当前目录下的 `input.txt` 中读取内容，然后把它输出。当执行 `./test` 命令，结果如图 5-11 所示。

结果提示 `Error opening file`，也就是读取文件时出现了异常，那究竟是什么异常呢，我们可以继续用 `strace` 来定位。执行 `strace ./test` 命令后，结果如图 5-12 所示（只截取了后面部分）。

```

[sharexu@linux 0503]$ ./test
Error opening file

```

图 5-11 例 5.3 程序执行结果图

```

mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9c4eb35000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9c4eb33000
arch_prctl(ARCH_SET_FS, 0x7f9c4eb33720) = 0
mprotect(0x7f9c4e175000, 16384, PROT_READ) = 0
mprotect(0x7f9c4e17000, 4096, PROT_READ) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9c4eb32000
mprotect(0x7f9c4e901000, 26572, PROT_READ) = 0
mprotect(0x7f9c4eb3e000, 4096, PROT_READ) = 0
munmap(0x7f9c4eb37000, 21000) = 0
brk(0) = 0xa08000
brk(0xa29000) = 0xa29000
open("input.txt", O_RDONLY) = -1 ENOENT (No such file or directory)
fcntl(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f9c4eb3c000
write(1, "Error opening file\n", 19)Error opening file
) = 19
exit_group(1) = ?
[sharexu@linux 0503]$

```

图 5-12 用 `strace` 定位程序读取文件发生的异常

其中 “`open("input.txt", O_RDONLY) = -1 ENOENT (No such file or directory)`”，是提示没有 `input.txt` 文件，再查看当前目录，得到如图 5-13 所示的结果。

```

[sharexu@linux 0503]$ ll
total 16
-rwxr-xr-x 1 sharexu user 10285 Oct 26 21:05 test
-rw-r--r-- 1 sharexu user 368 Oct 26 21:05 test.cpp
[sharexu@linux 0503]$

```

图 5-13 查看当前目录的结果图

图 5-13 展示当前目录只有 `test` 和 `test.cpp` 两个文件，没有 `input.txt` 文件。有时候，当你百思不得其解的时候，`strace` 往往是最佳选择，它能让你思路焕然一新。

5.2 gdb

`gdb` 是 `gcc` 的调试工具，主要用于 C 和 C++ 这两种语言编写的程序。它的功能很强大，主要体现在以下 4 点：①启动程序，可以按照用户自定义的要求随心所欲地运行程序，②可以让被调试的程序在指定的断点处停住；③当程序被停住时，可以检查此时程序中运行的状态；④动态地改变程序的执行环境。

要调试 C/C++ 的程序，首先在编译时，必须要把调试信息加到可执行文件中。使用编译器 (cc/gcc/g++) 的 -g 参数可以做到这一点，如下代码：

```
gcc -g hello.c -o hello
g++ -g hello.cpp -o hello
```

如果没有 -g，你将看不见程序的函数名、变量名，所代替的全是运行时的内存地址。当用 -g 把调试信息加入之后，并成功编译目标代码以后，让我们来看看如何用 gdb 来调试它。

启动 gdb 的方法：

1) gdb program

program 也就是你的执行文件，一般在当前目录下。

2) gdb program core

用 gdb 同时调试一个运行程序和 core 文件，core 是程序非法执行后 core dump 后产生的文件。

3) gdb program 1234

如果程序是一个服务程序，那么可以指定这个服务程序运行时的进程 ID，gdb 会自动进行 attach 操作，并调试这个程序。并且 program 应该在 PATH 环境变量中搜索得到。

1. gdb 常用用法

接下来将结合一个例子来看 gdb 的常用用法。

【例 5.4】斐波那契例子。

```
#include<iostream>
using namespace std;
int func(int n){
    int result=0;
    for(int i=1;i<=n;i++){
        result+=i;
    }
    return result;
}
int main(){
    int arr[10];
    arr[0]=0;
    arr[1]=1;
    for(int i=2;i<10;i++){
        arr[i]=arr[i-1]+arr[i-2];
    }
    cout<<"arr[9]"<<arr[9]<<endl;
    cout<<"func(9)"<<func(9)<<endl;
    return 0;
}
```

用 `g++ -g -o test test.cpp` 命令编译程序，注意这里要加上 `-g`。程序的执行结果如图 5-14 所示。

接下来用 `gdb` 调试程序，输入 `gdb test` 命令，启动 `gdb`。执行结果如图 5-15 所示。

输入“1”后（1 命令相当于 `list`），从第一行开始列出源码，结果如图 5-16 所示。

```
[sharexu@linux 0504]$ ./test
arr[9]34
func(9)45
```

图 5-14 例 5.4 程序的执行结果

```
[sharexu@linux 0504]$ gdb test
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/sharexu/chapter05/0504/test...done.
(gdb) []
```

图 5-15 开始用 `gdb` 调试程序时会输出的结果

再按下 `Enter` 键，表示重复上一次命令，结果如图 5-17 所示。

```
(gdb) l
3      int func(int n){
4          int result=0;
5          for(int i=1;i<=n;i++){
6              result+=i;
7          }
8          return result;
9      }
10     int main(){
11         int arr[10];
12         arr[0]=0;
(gdb) []
```

图 5-16 在 `gdb` 调试过程中输入 1 时展示的结果

```
(gdb)
13     arr[i]=1;
14     for(int i=2;i<10;i++){
15         arr[i]=arr[i-1]+arr[i-2];
16     }
17     cout<<"arr[9]"<<arr[9]<<endl;
18     cout<<"func(9)"<<func(9)<<endl;
19     return 0;
20 }
(gdb) []
```

图 5-17 在 `gdb` 调试过程中按下 `Enter` 键时展示的结果

执行“`b 15`”，表示设置在源码 15 行处设置断点，执行“`b func`”，表示设置断点在函数 `func` 入口处，执行“`info break`”，表示查看断点的信息，如图 5-18 所示。

```
(gdb) b 15
Breakpoint 1 at 0x4008a7: file test.cpp, line 15.
(gdb) b func
Breakpoint 2 at 0x40085b: file test.cpp, line 4.
(gdb) info break
Num   Type             Disp Enb Address            What
1     breakpoint        keep y  0x00000000004008a7 in main() at test.cpp:15
2     breakpoint        keep y  0x000000000040085b in func(int) at test.cpp:4
(gdb) []
```

图 5-18 在 `gdb` 调试过程中设置断点

执行 `r` 命令，表示运行程序，`run` 命令简写，如图 5-19 所示。

```
(gdb) r
Starting program: /home/sharexu/chapter05/0504/test

Breakpoint 1, main () at test.cpp:15
15     arr[i]=arr[i-1]+arr[i-2];
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.149.el6_6.5.x86_64 libgcc-4.4.6-4.el6.x86_64 libstdc++-4.4.6-4.el6.x86_64
(gdb) []
```

图 5-19 在 `gdb` 调试过程中运行程序

图 5-19 表示, 程序停在了断点处。

输入“n”, 表示单条语句执行, next 命令简写, 如图 5-20 所示。

输入“p i”“p arr[i]”, 分别打印变量 i 和变量 arr[i] 的值, 如图 5-21 所示。

输入“bt”, 查看函数堆栈, 如图 5-22 所示。

```
(gdb) n
14      for(int i=2;i<10;i++){
(gdb) □
```

图 5-20 在 gdb 调试过程中单条执行程序

```
(gdb) n
14      for(int i=2;i<10;i++){
(gdb) p i
$1 = 2
(gdb) p arr[i]
$2 = 1
(gdb) □
```

图 5-21 在 gdb 调试过程中打印变量的值

```
17      cout<<"arr[9]"<<arr[9]<<endl;
(gdb) arr[9]34
18      cout<<"func(9)"<<func(9)<<endl;
(gdb) n

Breakpoint 2. func (n=9) at test.cpp:4
4      int result=0;
(gdb)
5      for(int i=1;i<=n;i++){
(gdb)
6          result+=i;
(gdb)
5      for(int i=1;i<=n;i++){
(gdb) bt
#0 func (n=9) at test.cpp:5
#1 0x000000000040090d in main () at test.cpp:18
(gdb) □
```

图 5-22 在 gdb 调试过程中查看函数堆栈

输入“finish”, 退出函数, 如图 5-23 所示。

程序结束时如图 5-24 所示。

```
(gdb) finish
Run till exit from #0 func (n=9) at test.cpp:5
0x000000000040090d in main () at test.cpp:18
18      cout<<"func(9)"<<func(9)<<endl;
Value returned is $3 = 45
(gdb) □
```

图 5-23 在 gdb 调试过程退出函数

```
(gdb) n
func(9)45
19      return 0;
(gdb)
20  }
(gdb)
0x00007ffff7c7d5d in __libc_start_main () from /lib64/libc.so.6
(gdb)
Single stepping until exit from function __libc_start_main,
which has no line number information.

Program exited normally.
(gdb) □
```

图 5-24 在 gdb 调试过程程序执行结束

输入“q”, 结束调试, 如图 5-25 所示。

上述一共用了以下这些命令。

l: 列出函数代码及其行数。

b 16: 在代码 16 行处设置断点。

b func: 在函数 func 处设置断点。

r: 运行程序。

n: 单条执行语句。

p i: 打印 i 变量的值。

bt: 查看函数的堆栈。

finish: 退出函数。

q: 结束调试。

```
Program exited normally.
(gdb) q
[sharexu@linux 0504]$ □
```

图 5-25 退出 gdb 调试状态

2. 用gdb分析coredump文件

gdb还可以用于分析coredump文件。core，又称之为coredump文件，是UNIX/Linux操作系统的一种机制，对于线上服务而言，core令人闻之色变，因为出core的过程意味着服务暂时不能正常响应，需要恢复，并且随着Core进程的内存空间越大，此过程可能持续很长一段时间（例如，当进程占用60GB+内存时，完整coredump文件需要15min左右才能完全写到磁盘上），这期间产生的流量损失，不可估量。

凡事皆有两面性，操作系统在coredump的同时，虽然会终止当前进程，但是也会保留下第一手的现场数据，操作系统仿佛是一架被按下快门的相机，而照片就是产出的coredump文件。coredump文件含有当进程被终止时内存、CPU寄存器和各种函数调用堆栈信息等，可以供后续开发人员进行调试。

（1）coredump文件的存储路径

有时候在执行程序时，会出现提示Segmentation fault，但在当前目录下却没有找到coredump文件，可以通过下面的命令看到core文件的存在位置：

```
cat /proc/sys/kernel/core_pattern
```

默认值是core，也就是当前目录，如果不是core，则是在指定的目录下。



注意 这里是指在进程当前工作目录的下创建。通常与程序在相同的路径下。但如果程序中调用了chdir函数，则有可能改变了当前工作目录。这时core文件创建在chdir指定的路径下。有好多程序即使崩溃了，也找不到core文件放在什么位置，这和chdir函数就有关系。当然程序崩溃了不一定都产生core文件。

通过下面的命令可以更改coredump文件的存储位置，若你希望把core文件生成到/data/coredump/core目录下：

```
echo "/data/coredump/core" > /proc/sys/kernel/core_pattern
```

注意，这里当前用户必须具有对/proc/sys/kernel/core_pattern的写权限。

默认情况下，内核在coredump时所产生的core文件放在与该程序相同的目录中，并且文件名固定为core。很显然，如果有多个程序产生core文件，或者同一个程序多次崩溃，就会重复覆盖同一个core文件，因此有必要对不同程序生成的core文件进行分别命名。

通过修改kernel的参数，可以指定内核所生成的coredump文件的文件名。例如，使用下面的命令使kernel生成名字为core.filename.pid格式的core dump文件：

```
echo "/data/coredump/core.%e.%p" > /proc/sys/kernel/core_pattern
```

这样配置后，产生的core文件中将带有崩溃的程序名、以及它的进程ID。上面的%e和%p会被替换成程序文件名以及进程的ID。

如果在上述文件名中包含目录分隔符“/”，那么所生成的core文件将会被放到指定的

目录中。需要说明的是,在内核中还有一个与 coredump 相关的设置,就是 /proc/sys/kernel/core_uses_pid。如果这个文件的内容被配置成 1,那么即使 core_pattern 中没有设置 %p,最后生成的 core dump 文件名仍会加上进程的 ID。

(2) 产生 coredump 文件的条件

有时候,程序 coredump 了,如图 5-26 所示,却没有生成 coredump 文件(执行程序时提示 Segmentation fault),那我们就该找找是什么原因导致没生成 coredump 文件。

1) 产生 coredump 文件的条件,首先需要确认当前会话的能生成的 coredump 文件大小,若大小为 0,则不会产生对应的 coredump 文件,这样就需要进行修改和设置了。ulimit -c 命令可以查看 coredump 文件大小的最大值。

```
[sharexu@linux 0506]$ ./test
Segmentation fault
```

图 5-26 程序 coredump 时的提示语

```
[sharexu@linux 0505]$ ulimit -c
0
```

图 5-27 系统不能生成 coredump 文件时执行 ulimit -c 命令的结果

执行 ulimit -c unlimited 命令,可以设置 coredump 文件的大小为不受限制,如图 5-28 所示。若想甚至对应的字符大小,则可以指定:

```
ulimit -c [size]
```

可以看出,这里的 size 的单位是 blocks,一般 1block=512Bytes

但当前设置的 ulimit 只对当前会话有效,若想系统均有效,则需要进行如下设置。

在 /etc/profile 中加入以下一行,这将允许生成 coredump 文件:

```
ulimit-c unlimited
```

2) 当前用户,即执行对应程序的用户具有对写入 core 目录的写权限以及有足够的空间。保证以上两点后,再执行文件,就会发现 coredump 时的提示变成了如图 5-29 所示的情况。

提示 Segmentation fault (core dumped),接着就只需去 cat /proc/sys/kernel/core_pattern 输出的目录下找到该 coredump 文件即可。

(3) 产生 coredump 文件的原因

造成程序 coredump 的原因有很多,这里总结一些比较常用的经验。

1) 内存访问越界,可能的具体原因是:①由于使用错误的下标,导致数组访问越界;②搜索字符串时,依靠字符串结束符来判断字符串是否结束,但是字符串没有正常的使用结束符;③使用 strcpy, strcat, sprintf, strcmp, strcasecmp 等字符串操作函数时,容易出现将目标字符串读/写越界的情况。应该使用 strncpy, strlcpy, strncat, strlcat, snprintf, strncmp,

```
[sharexu@linux 0505]$ ulimit -c
0
[sharexu@linux 0505]$ ulimit -c unlimited
[sharexu@linux 0505]$ ulimit -c
unlimited
[sharexu@linux 0505]$
```

图 5-28 修改系统可生成 coredump 文件大小限制的命令

```
[sharexu@linux 0506]$ ./test
Segmentation fault (core dumped)
```

图 5-29 有 coredump 文件产生时的提示语

strncasecmp 等函数防止读容易出现写越界。

2) 多线程程序使用了线程不安全的函数。

应该使用下面这些可重入的函数，它们很容易被用错：asctime_r(3c)、gethostbyname_r(3n)、getservbyname_r(3n)、ctermid_r(3s)、gethostent_r(3n)、getservbyport_r(3n)、ctime_r(3c)、getlogin_r(3c)、getservent_r(3n)、fgetgrent_r(3c)、getnetbyaddr_r(3n)、getspent_r(3c)、fgetpwent_r(3c)、getnetbyname_r(3n)、getspnam_r(3c)、fgetspent_r(3c)、getnetent_r(3n)、gmtime_r(3c)、gamma_r(3m)、getnetgrent_r(3n)、lgamma_r(3m)、getauclassent_r(3)、getprotobyname_r(3n)、localtime_r(3c)、getauclassnam_r(3)、etprotobyname_r(3n)、nis_sperror_r(3n)、getauevent_r(3)、getprotoent_r(3n)、rand_r(3c)、getauenvnam_r(3)、getpwent_r(3c)、readdir_r(3c)、getauenvnum_r(3)、getpwnam_r(3c)、strtok_r(3c)、getgrent_r(3c)、getpwuid_r(3c)、tmpnam_r(3s)、getgrgid_r(3c)、getrpcbyname_r(3n)、ttyname_r(3c)、getgrnam_r(3c)、getrpcbynumber_r(3n)、gethostbyaddr_r(3n) 和 getrpcent_r(3n)。

3) 多线程读写的数据未加锁保护。

对于会被多个线程同时访问的全局数据，应该注意加锁保护，否则很容易造成 coredump。

4) 非法指针，包括使用空指针或随意使用指针转换。

随意使用指针转换是指一个指向一段内存的指针，除非确定这段内存原先就分配为某种结构或类型，或者这种结构或类型的数组，否则不要将它转换为这种结构或类型的指针，而应该将这段内存拷贝到一个这种结构或类型中，再访问这个结构或类型。这是因为如果这段内存的开始地址不是按照这种结构或类型对齐的，那么访问它时就很容易因为 bus error 出现 core dump 错误。

5) 堆栈溢出。

不要使用大的局部变量（因为局部变量都分配在栈上），这样容易造成堆栈溢出，破坏系统的栈和堆结构，导致出现莫名其妙的错误。

3. gdb 定位 coredump 文件

【例 5.6】非法访问内存。

```
#include <stdio.h>
int main(){
    int b=1;
    int* a;
    *a=b;
    return 0;
}
```

用 g++ -g -o test test.cpp 编译该文件，得到 test 这个可执行文件。执行 ./test 命令后，结果如图 5-30 所示。

对程序进行 coredump 后，把 coredump 文件放到当前目录下以备分析，先来看下该 coredump 文件的 ELF 头部，如图 5-31 所示。

```
[sharexu@linux 0506]$ ./test
Segmentation fault (core dumped)
```

图 5-30 例 5.6 程序的执行结果

```
[sharexu@linux 0506]$ readelf -h core.test.13093
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:             ELF64
  Data:              2's complement, little endian
  Version:           1 (current)
  OS/ABI:            UNIX - System V
  ABI Version:       0
  Type:              CORE (Core file)
  Machine:           Advanced Micro Devices X86-64
  Version:           0x1
  Entry point address: 0x0
  Start of program headers: 64 (bytes into file)
  Start of section headers: 0 (bytes into file)
  Flags:             0x0
  Size of this header: 64 (bytes)
  Size of program headers: 56 (bytes)
  Number of program headers: 29
  Size of section headers: 0 (bytes)
  Number of section headers: 0
  Section header string table index: 0
[sharexu@linux 0506]$
```

图 5-31 用 readelf 命令查看 coredump 文件的 ELF 头部

可以看到文件类型是 CORE，表示这是 core-dump 文件。

使用 GDB，先从可执行文件中读取符号表信息，然后读取 core 文件。如果不与可执行文件一起操作可以吗？答案是不行的，因为 core 文件中没有符号表信息，无法进行调试，可以使用如下命令来验证：

```
objdump -x core.test.13093 | tail
```

执行结果如图 5-32 所示。

其中可以看到如下两行信息：

```
SYMBOL TABLE:
no symbols
```

```
[sharexu@linux 0506]$ objdump -x core.test.13093 | tail
36 load26      00016000 00007ffff4854000 0000000000000000 0003c000 2**12
                CONTENTS, ALLOC, LOAD
37 load27      00001000 00007ffff489a000 0000000000000000 00052000 2**12
                CONTENTS, ALLOC, LOAD, READONLY, CODE
38 load28      00009000 ffffffff600000 0000000000000000 00053000 2**12
                ALLOC, READONLY, CODE
SYMBOL TABLE:
no symbols
[sharexu@linux 0506]$
```

图 5-32 用 objdump 命令查看 coredump 文件的符号表

表明当前的 ELF 格式文件中没有符号表信息。

接着来看下是 core 在哪了。执行 gdb test core.test.13093 命令后结果如图 5-33 所示。

```
[sharexu@linux 0506]$ gdb test core.test.13093
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/sharexu/chapter05/0506/test...done.
[New Thread 13093]
Missing separate debuginfo for
Try: yum --disablerepo="*" --enablerepo="*" debuginfo install /usr/lib/debug/.build-id/80/1b9600daa2cd5f7035
ad415e9c7dd0e6bb0a2
Reading symbols from /usr/lib64/libstdc++.so.6...(no debugging symbols found)...done.
Loaded symbols for /usr/lib64/libstdc++.so.6
Reading symbols from /lib64/libm.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib64/libm.so.6
Reading symbols from /lib64/libgcc_s.so.1...(no debugging symbols found)...done.
Loaded symbols for /lib64/libgcc_s.so.1
Reading symbols from /lib64/libc.so.6...(no debugging symbols found)...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
Core was generated by ./test.
Program terminated with signal 11, Segmentation fault.
#0 0x0000000000000000 in main () at test.cpp:5
5      *a=b;
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.149.el6_6.5 x86_64 libgcc-4.4.6-4.el6.x86_64 libstdc++-4.4.6-4.el6.x86_64
(gdb)
```

图 5-33 用 gdb 调试 coredump 文件

结果显示是在程序第 5 行，*a=b 处，分别打印变量 a 和变量 b 的值，如图 5-34 所示。

发现 a 变量指向的地址是非法区域，也就是因为 a 没有分配内存导致。第 5 行应该改成：

```
a=&b;
```

这样就把 a 指向 b 了。

可以使用 GDB 一步一步地调试程序，也可以用 GDB 调试 coredump 的程序。建议读者把命令都熟记，这样用起来会更加得心应手。

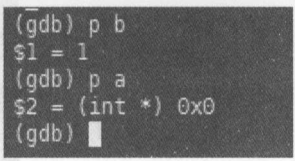


图 5-34 在 gdb 调试过程中打印变量的值

5.3 top

top 命令是 Linux 下常用的性能分析工具，能够实时显示系统中各个进程的资源占用状况，类似于 Windows 的任务管理器。下面详细介绍它的使用方法。

top 命令的部分截图如图 5-35 所示。

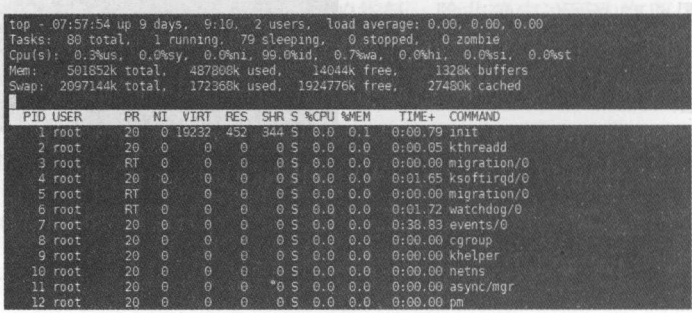


图 5-35 top 命令结果截图

第 1 行分别显示：系统当前时间、系统运行时间、当前用户登录数和系统负载。系统负载 (load average)，这里有 3 个数值，分别是系统最近 1min、5min、15min 的平均负载。一般对于单个处理器来说，负载在 0~1.00 是正常的，超过 1.00 就要引起注意了。在多核处理器中，系统均值不应该高于处理器核心的总数。

第 2 行分别显示：total(进程总数)、running(正在运行的进程数)、sleeping(睡眠的进程数)、stopped(停止的进程数)和 zombie(僵尸进程数)。

第 3 行分别显示：%us(用户空间占用 CPU 百分比)、%sy(内核空间占用 CPU 百分比)、%ni(用户进程空间内改变过优先级的进程占用 CPU 百分比)、%id(空闲 CPU 百分比)、%wa(等待输入输出(I/O)的 CPU 时间百分比)、%hi(cpu 处理硬件中断的时间)、%si(cpu 处理软中断的时间)、%st(用于有虚拟 cpu 的情况)。通常 id% 值可以反映一个系统 cpu 的闲忙程度。

第 4 行则显示内存 MEM 的数据：total(物理内存总量)、used(使用的物理内存总量)、

free (空闲内存总量)、buffers (用作内核缓存的内存量)。

第 5 行则显示交换器 SWAP 的数据: total(交换区总量)、used(使用的交换区总量)、free(空闲交换区总量)、cached (缓冲的交换区总量)。

buffers 和 cached 的区别需要说明一下, buffers 指的是块设备的读写缓冲区, cached 指的是文件系统本身的页面缓存。它们都是 Linux 操作系统底层的机制, 目的就是为了加速对磁盘的访问。

第 6 行则显示 PID (进程号)、USER (运行用户)、PR (优先级)、NI (任务 nice 值)、VIRT (虚拟内存用量)VIRT=SWAP+RES、RES(物理内存用量)、SHR(共享内存用量)、S(进程状态)、%CPU (CPU 占用比)、%MEM (物理内存占用比)、TIME+ (累计 CPU 占用时间)、COMMAND 命令名 / 命令行。

top 命令显示系统当前的进程和其他状况, top 是一个动态显示过程, 即可以通过用户按键来不断刷新当前状态。如果在前台执行该命令, 它将独占前台, 直到用户终止该程序为止。比较准确地说, top 命令提供了实时地对系统处理器的状态监视。它将显示系统中 CPU 最“敏感”的任务列表。该命令可以按 CPU 使用、内存使用和执行时间对任务进行排序; 而且该命令的很多特性都可以通过交互式命令或者在个人定制文件中进行设定。

输入“q”, 则退出 top 命令。

5.4 ps

Linux 中的 ps (process status) 命令列出的是当前在运行的进程的快照, 就是执行 ps 命令的那个时刻的那些进程, 如果想要动态地显示进程信息, 就可以使用 top 命令。

要对进程进行监测和控制, 首先必须要了解当前进程的情况, 也就是需要查看当前进程, 而 ps 命令就是最基本同时也是非常强大的进程查看命令。使用该命令可以确定有哪些进程正在运行及其运行的状态、进程是否结束、进程有没有僵死、哪些进程占用了过多的资源等。总之大部分信息都是可以通过执行该命令得到的。

ps 命令提供进程的一次性的查看, 它所提供的查看结果并不动态连续的; 如果想对进程时间监控, 应该用 top 命令。

kill 命令用于杀死进程。

(1) Linux 上进程有 5 种状态, 如下所述。

- 1) 运行 (正在运行或在运行队列中等待)。
- 2) 中断 (休眠中, 受阻, 在等待某个条件的形成或接受到信号)。
- 3) 不可中断 (收到信号不唤醒和不可运行, 进程必须等待直到有中断发生)。
- 4) 僵死 (进程已终止, 但进程描述符存在, 直到父进程调用 wait4() 系统调用后释放)。
- 5) 停止 (进程收到 SIGSTOP, SIGSTP, SIGTIN, SIGTOU 信号后停止运行运行)。

(2) ps 工具标识进程的 5 种状态码, 如下所述。

- 1) D 不可中断: uninterruptible sleep (usually IO)。
- 2) R 运行: runnable (on run queue)。
- 3) S 中断: sleeping。
- 4) T 停止: traced or stopped。
- 5) Z 僵死: a defunct ("zombie") process。

命令格式是: ps[参数]。命令功能是用来显示当前进程的状态。

1. ps 命令常用参数

表 5-1 描述了 ps 命令的常用参数。

2. ps 命令使用实例

这里,我们先写一个程序,使其能运行一段时间,方便我们更好地了解 ps 命令。

【例 5.7】 ps 命令学习实例。

```
#include<iostream>
using namespace std;
int main(){
    for(int i=0;i<100;i++){
        cout<<"i:"<<i<<endl;
        sleep(5);
    }
    return 0;
}
```

程序的执行结果如图 5-36 所示。

例 5.7 中,是利用 for 循环打印一个数字,每打一次就休眠 5s,一共打印 100 次。

(1) 显示指定用户信息。

命令: ps -u sharexu

输出: 如图 5-37 所示。

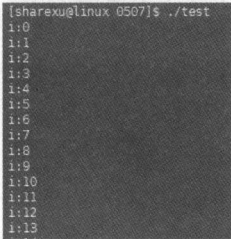


图 5-36 例 5.7 程序的执行结果
(省略部分结果)

表 5-1 ps 命令常用参数	
参 数	功 能
a	显示所有进程
-a	显示同一终端下的所有程序
-A	显示所有进程
c	显示进程的真实名称
-e	-e 等于“-A”
e	显示环境变量
f	显示程序间的关系
-H	显示树状结构
r	显示当前终端的进程
T	显示当前终端的所有程序
u	指定用户的所有进程
-au	显示较详细的资讯
-aux	显示所有包含其他使用者的行程
-C	列出指定命令的状况
--help	显示帮助信息
--version	显示版本

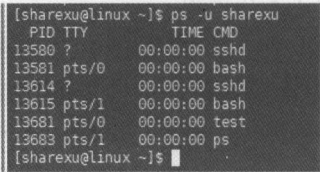


图 5-37 用 ps 命令显式指定用户的信息

其中的 test 进程,正是刚刚手动运行的那个。

(2) 显示所有进程信息，连同命令行。

命令：ps -ef

输出：如图 5-38 所示。

(3) ps 与 grep 常用组合用法，查找特定进程。

命令：ps -ef|grep test

输出：如图 5-39 所示。

```
[sharexu@linux ~]$ ps -ef
UID          PID    PPID  C  STATE TTY          TIME CMD
root         1        0  0   Oct17 ?        00:00:00 /sbin/init
root         2        0  0   Oct17 ?        00:00:00 [kthreadd]
root         3        2  0   Oct17 ?        00:00:00 [migration/0]
root         4        2  0   Oct17 ?        00:00:00 [kscotlrqd/0]
root         5        2  0   Oct17 ?        00:00:00 [migration/0]
root         6        2  0   Oct17 ?        00:00:00 [watchdog/0]
root         7        2  0   Oct17 ?        00:00:41 [events/0]
```

图 5-38 用 ps 命令显式所有进程信息
(省略部分结果)

```
[sharexu@linux ~]$ ps -ef | grep test
sharexu 13681 13581 0 21:08 pts/0    00:00:00 ./test
sharexu 13690 13615 0 21:12 pts/1    00:00:00 grep test
[sharexu@linux ~]$
```

图 5-39 结合 ps 命令和 grep 命令查找特定进程

(4) 将目前登入的 PID 与相关信息列示出来。

命令：ps -l

输出：如图 5-40 所示。

```
[sharexu@linux ~]$ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  506 13615 13614 0 80   0 - 27076 wait  pts/1    00:00:00 bash
0 R  506 13691 13615 0 80   0 - 27031 -    pts/1    00:00:00 ps
[sharexu@linux ~]$
```

图 5-40 用 ps 命令显式与这次登入的 PID 的相关信息

各相关信息的意义如下所述。

1) F 代表这个程序的旗标 (flag)，4 代表使用者为 super user。

2) S 代表这个程序的状态 (STAT)，关于各 STAT 的意义将在下文中介绍。

3) UID 程序被该 UID 所拥有。

4) PID 就是这个程序的进程 id。

5) PPID 则是其父进程的进程 id。

6) C 是使用的 CPU 资源百分比。

7) PRI 是 Priority (优先执行序) 的缩写。

8) NI 是 Nice 值。

9) ADDR 是 kernel function，指出该程序在内存的那个部分。如果是个 running 的程序，一般就是 "-"。

10) SZ 使用掉的内存大小。

11) WCHAN 目前这个程序是否正在运作当中，若为 - 表示正在运作。

12) TTY 登入者的终端机位置。

13) TIME 使用掉的 CPU 时间。

14) CMD 所下达的指令内容。

在预设的情况下，ps 仅会列出与目前所在的 bash shell 有关的 PID 而已，所以，当使用 ps -l 的时候，只有 2 个 PID。

(5) 列出目前所有的正在内存当中的程序。

命令：ps aux

输出：如图 5-41 所示。

```
[sharexu@linux ~]$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0 19232  268 ?        Ss   Oct17  0:00 /sbin/init
root         2  0.0  0.0      0   0 ?        S   Oct17  0:00 [kthreadd]
root         3  0.0  0.0      0   0 ?        S   Oct17  0:00 [migration/0]
root         4  0.0  0.0      0   0 ?        S   Oct17  0:01 [ksoftirqd/0]
root         5  0.0  0.0      0   0 ?        S   Oct17  0:00 [migration/0]
root         6  0.0  0.0      0   0 ?        S   Oct17  0:01 [watchdog/0]
root         7  0.0  0.0      0   0 ?        S   Oct17  0:41 [events/0]
root         8  0.0  0.0      0   0 ?        S   Oct17  0:00 [cgrou]
root         9  0.0  0.0      0   0 ?        S   Oct17  0:00 [khelper]
```

图 5-41 用 ps 命令列出目前所有的正在内存当中的程序（省略部分结果）

- 1) USER：该进程属于那个使用者账号的。
- 2) PID：该进程的号码。
- 3) %CPU：该进程使用掉的 CPU 资源百分比。
- 4) %MEM：该进程所占用的物理内存百分比。
- 5) VSZ：该进程使用掉的虚拟内存量 (KBytes)。
- 6) RSS：该进程占用的固定的内存量 (KBytes)。
- 7) TTY：该进程是在那个终端机上面运作，若与终端机无关，则显示“?”，另外，tty1-tty6 是本机上面的登入者程序，若为 pts/0 等，则表示为由网络连接进主机的程序。
- 8) STAT：该程序目前的状态，主要的状态有以下几种。
 - a.R：该程序目前正在运作，或者是可被运作。
 - b.S：该程序目前正在睡眠当中（可说是 idle 状态），但可被某些信号 (signal) 唤醒。
 - c.T：该程序目前正在侦测或者是停止了。
 - d.Z：该程序应该已经终止，但是其父程序却无法正常地终止它，造成 zombie (僵死) 程序的状态。
 - e.START：该 process 被触发启动的时间。
 - f.TIME：该 process 实际使用 CPU 运作的时间。
 - g.COMMAND：该程序的实际指令。

关于进程和进程的状态，后面会有详细的章节专门来讲，这里就先不展开了。

5.5 Valgrind

5.5.1 Valgrind 概述

接下来再来看一款内存分析工具 Valgrind。Valgrind 是一套 Linux 下的开放源代码的仿

真调试工具的集合。Valgrind 由内核以及基于内核的其他调试工具组成。内核类似于一个框架，它模拟了一个 CPU 环境，并提供服务给其他工具；而其他工具则类似于插件，利用内核提供的服务完成各种特定的内存调试任务。Valgrind 的体系结构如图 5-42 所示。

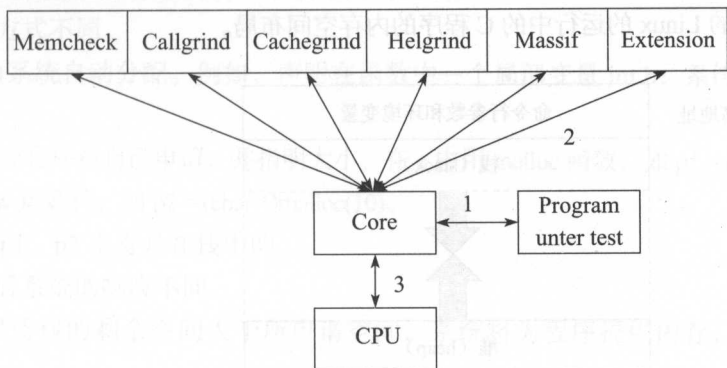


图 5-42 Valgrind 体系结构图

Valgrind 包括如下一些工具。

(1) Memcheck：这是 Valgrind 应用最广泛的工具，一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。这些问题往往是 C/C++ 程序员最头疼的问题，Memcheck 在这里帮上了大忙。

(2) Callgrind：和 gprof 类似的分析工具，但它对程序的运行观察更是入微，能提供更多的信息。和 gprof 不同，它不需要在编译源代码时附加特殊选项，但推荐加上调试选项。Callgrind 收集程序运行时的一些数据，建立函数调用关系图，还可以有选择地进行 Cache 模拟。在运行结束时，它会把分析数据写入一个文件。callgrind_annotate 可以把这个文件的内容转化成可读的形式。

(3) Cachegrind：它主要用来检查程序中缓存使用出现的问题。Cache 分析器，它模拟 CPU 中的一级缓存 I1、D1 和二级缓存，能够精确地指出程序中 Cache 的丢失和命中。如果需要，它还能够为用户提供 Cache 丢失次数、内存引用次数以及每行代码、每个函数、每个模块及整个程序产生的指令数，这对优化程序有很大的帮助。

(4) Helgrind：它主要用来检查多线程程序中出现的竞争问题。Helgrind 寻找内存中被多个线程访问，而又没有一贯加锁的区域，这些区域往往是线程之间失去同步的地方，而且会导致难以发掘的错误。Helgrind 实现了名为 Eraser 的竞争检测算法，并做了进一步改进，减少了报告错误的次数。

(5) Massif：堆栈分析器，它能测量程序在堆栈中使用了多少内存，告诉我们堆块、堆管理块和栈的大小。Massif 能帮助我们减少内存的使用，在带有虚拟内存的现代系统中，它还能够加速程序的运行，减少程序停留在交换区中的几率。

(6) Extension：可以利用 Core 提供的功能，自己编写特定的内存调试工具。

5.5.2 Linux 程序内存空间布局

1. 程序内存空间布局概述

要发现 Linux 下的内存问题，首先一定要知道在 Linux 下，内存是如何被分配的。图 5-43 展示了一个典型的 Linux 的运行中的 C 程序的内存空间布局。

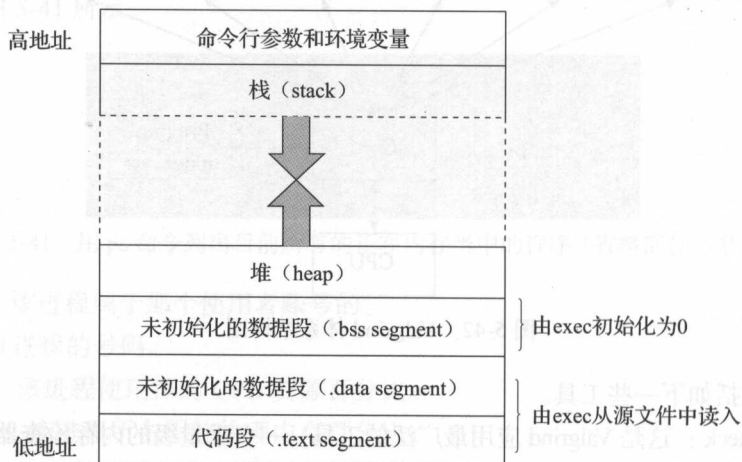


图 5-43 典型内存空间布局

一个典型的 Linux 下的 C 程序内存空间由如下几部分组成。

（1）代码段（.text segment）：代码段通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。程序段是程序代码在内存中的映射，一个程序可以在内存中有多个副本。

（2）初始化数据段（.data segment）：通常是指用来存放程序中已初始化的全局变量的一块内存区域，例如，位于所有函数之外的全局变量：`int val=100`。需要强调的是，以上内容都是位于程序的可执行文件中，内核在调用 `exec` 函数启动该程序时从源程序文件中读入。数据段属于静态内存分配。

（3）未初始化数据段（.bss segment）：通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS 是 Block Started by Symbol 的简称。

（4）堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态地扩张或缩减。当进程调用 `malloc/free` 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）或释放的内存从堆中被剔除（堆被缩减）。

（5）栈（stack）：栈又称堆栈，存放程序的局部变量（但不包括 `static` 声明的变量，`static` 意味着在数据段中存放变量）。除此以外，在函数被调用时，栈用来传递参数和返回值。由于栈的先进后出特点，所以栈特别方便用来保存 / 恢复调用现场。而动态内存分配，需要程

序员手工分配，手工释放。

2. 堆栈的区别

由于堆、栈尤为重要，这里讲下堆栈之间的区别。

(1) 申请方式不同。

1) 栈：由系统自动分配。例如，声明在函数中一个局部变量 `int b`；系统自动在栈中为 `b` 开辟空间。

2) 堆：需要程序员自己申请，并指明大小，在 `c` 中用 `malloc` 函数，如 `p1=(char *)malloc(10)`；在 `C++` 中用 `new` 运算符，如 `p2=(char *)malloc(10)`。

但是注意 `p1`、`p2` 本身是在栈中的。

(2) 申请后系统的响应不同。

1) 栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常，提示栈溢出。

2) 堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间中大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。其次，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样代码中的 `delete` 语句才能正确的释放本内存空间。最后，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动地将多余的那部分重新放入空闲链表中。

(3) 申请大小的限制不同。

1) 栈：栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 `Linux` 下，栈的大小是一个常数（虽然可以设置，但它是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示 `overflow`。因此，能从栈获得的空间较小。用 `ulimit -a` 命令可以看到栈大小的限制，如图 5-44 所示，其中 `stack size` 的值就是栈的大小了（本测试机是 10M）。可以通过 `ulimit -s` 修改栈的大小，。

2) 堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

(4) 申请效率不同。

1) 栈由系统自动分配，速度较快；但程序员是无法控制的。

```
[sharexu@linux 0514]$ ulimit -l
-bash: ulimit: -l: invalid option
ulimit: usage: ulimit [-SHacdeflmnpqrstuvx] [limit]
[sharexu@linux 0514]$ ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 3790
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files              (-n) 65535
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 10240
cpu time                (seconds, -t) unlimited
max user processes      (-u) 1024
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
[sharexu@linux 0514]$
```

图 5-44 `ulimit -a` 命令查看栈大小限制

2) 堆是由 new 分配的内存, 一般速度比较慢, 而且容易产生内存碎片; 不过用起来最方便。

(5) 堆和栈中的存储内容不同。

1) 栈: 在函数调用时, 第一个进栈的是主函数中后的下一条指令(函数调用语句的下一条可执行语句)的地址, 然后是函数的各个参数, 在大多数的 C 编译器中, 参数是由右往左入栈的, 然后是函数中的局部变量。注意静态变量是不入栈的。当本次函数调用结束后, 局部变量先出栈, 然后是参数, 最后栈顶指针指向最开始存的地址, 也就是主函数中的下一条指令, 程序由该点继续运行。

2) 堆: 一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容 by 程序员安排。

堆和栈的区别可以用如下比喻来形象看出。

使用栈就像我们去饭馆里吃饭, 只管点菜(发出申请)、付钱和吃(使用), 吃饱了就走, 不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作, 它的好处是快捷, 但是自由度小。

使用堆就像是自己动手做喜欢吃的菜肴, 比较麻烦, 但是比较符合自己的口味, 而且自由度大。

虽然堆栈的说法现在趋向于变成一个整体, 但是它们还是有很大区别的, 连着说只是由于历史的原因。

3. 实例讲内存空间布局。

接下来用一个例子来说明内存空间布局。

【例 5.8】变量与函数地址分析。

```
#include<stdio.h>
#include<stdlib.h>
int g1=0, g2=0, g3=0;
int max(int i)
{
    int m1=0,m2,m3=0,*p_max;
    static int n1_max=0,n2_max,n3_max=0;
    p_max = (int*)malloc(10);
    printf(" 打印 max 程序地址 \n");
    printf("in max: %x\n\n",max);
    printf(" 打印 max 传入参数地址 \n");
    printf("in max: %x\n\n",&i);
    printf(" 打印 max 函数中静态变量地址 \n");
    printf("%x\n",&n1_max);      // 打印各本地变量的内存地址
    printf("%x\n",&n2_max);
    printf("%x\n\n",&n3_max);
    printf(" 打印 max 函数中局部变量地址 \n");
    printf("%x\n",&m1);          // 打印各本地变量的内存地址
    printf("%x\n",&m2);
    printf("%x\n\n",&m3);
```



```

printf(" 打印 max 函数中 malloc 分配地址 \n");
printf("%x\n\n",p_max); // 打印各本地变量的内存地址
if(i) return 1;
else return 0;
}

int main(int argc, char **argv)
{
    static int s1=0, s2, s3=0;
    int v1=0, v2, v3=0;
    int *p;
    p = (int*)malloc(10);
    printf(" 打印各全局变量 (已初始化) 的内存地址 \n");
    printf("%x\n",&g1); // 打印各全局变量的内存地址
    printf("%x\n",&g2);
    printf("%x\n\n",&g3);
    printf("=====\n");
    printf(" 打印程序初始程序 main 地址 \n");
    printf("main: %x\n\n", main);
    printf(" 打印主参地址 \n");
    printf("argv: %x\n\n",argv);
    printf(" 打印各静态变量的内存地址 \n");
    printf("%x\n",&s1); // 打印各静态变量的内存地址
    printf("%x\n",&s2);
    printf("%x\n\n",&s3);
    printf(" 打印各局部变量的内存地址 \n");
    printf("%x\n",&v1); // 打印各本地变量的内存地址
    printf("%x\n",&v2);
    printf("%x\n\n",&v3);
    printf(" 打印 malloc 分配的堆地址 \n");
    printf("malloc: %x\n\n",p);
    printf("=====\n");
    max(v1);
    printf("=====\n");
    printf(" 打印子函数起始地址 \n");
    printf("max: %x\n\n",max);
    return 0;
}

```

程序的执行结果如下:

打印各全局变量 (已初始化) 的内存地址

600f40

600f44

600f48

=====

打印程序初始程序 main 地址

main: 400760

打印主参地址

argv: 64857198


```
打印各静态变量的内存地址
600f4c
600f50
600f54

打印各局部变量的内存地址
648570a4
648570a0
6485709c

打印 malloc 分配的堆地址
malloc: 1da6010

=====
打印 max 程序地址
in max: 400634

打印 max 传入参数地址
in max: 6485704c

打印 max 函数中静态变量地址
600f58
600f5c
600f60

打印 max 函数中局部变量地址
64857064
64857060
6485705c

打印 max 函数中 malloc 分配地址
1da6030

=====
打印子函数起始地址
max: 400634
```

接着整理程序的运行结果，如表 5-2 所示。

表 5-2 变量与函数地址布局

说 明	地 址
main 函数中局部变量地址	648570a4
main 函数中局部变量地址	648570a0
main 函数中局部变量地址	6485709c
max 函数中局部变量地址	64857064
max 函数中局部变量地址	64857060
max 函数中局部变量地址	6485705c

(续)

说 明	地 址
max 传入参数地址	6485704c
max 函数中 malloc 分配的堆地址	1da6030
main 函数中 malloc 分配的堆地址	1da6010
max 函数中静态变量地址	600f60
max 函数中静态变量地址	600f5c
max 函数中静态变量地址	600f58
main 函数中静态变量地址	600f54
main 函数中静态变量地址	600f50
main 函数中静态变量地址	600f4c
全局变量地址	600f48
全局变量地址	600f44
全局变量地址	600f40
main 函数地址	400760
max 函数地址	400634

可以大致查看整个程序在内存中的分配情况，有以下 2 个特点。

(1) 传入的参数，局部变量，都是在栈顶分布，随着子函数的增多而向下增长。

(2) 函数的调用地址（函数运行代码），全局变量，静态变量都是在分配内存的底部存在，而 malloc 分配的堆则存在于这些内存之上，并向上生长。

5.5.3 内存检查原理

Memcheck 检测内存问题的原理如图 5-45 所示。

Memcheck 能够检测出内存问题，关键在于其建立了两个全局表，如下所述。

(1) Valid-Value 表：对于进程的整个地址空间中的每一个字节（Byte），都有与之对应的 8 bit；对于 CPU 的每个寄存器，也有一个与之对应的 bit 向量。这些 bit 负责记录该字节或者寄存器值是否具有有效的、已初始化的值。

(2) Valid-Address 表：对于进程整个地址空间中的每一个字节（Byte），还有与之对应的 1 bit，负责记录该地址是否能够被读写。

检测原理：当要读写内存中某个字节时，首先检查这个字节对应的 A bit。如果该 A bit 显示该位置是无效位置，Memcheck 则报告读写错误。

内核（core）类似于一个虚拟的 CPU 环境，这样当内存中的某个字节被加载到真实的 CPU 中时，该字节对应的 V bit 也被加载到虚拟的 CPU 环境中。一旦寄存器中的值，被用来产生内存地址，或者该值能够影响程序输出，则 Memcheck 会检查对应的 V bit，如果该值尚未初始化，则会报告使用未初始化内存错误。

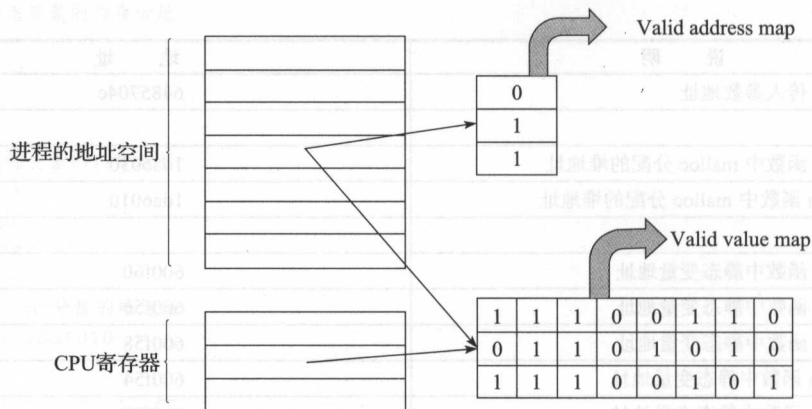


图 5-45 Memcheck 内存检查原理图

5.5.4 Valgrind 安装

Valgrind 的官网地址是：<http://www.valgrind.org>

执行以下命令即可安装成功：

```
wget http://www.valgrind.org/downloads/valgrind-3.8.1.tar.bz2
tar xvf valgrind-3.8.1.tar.bz2
cd valgrind-3.8.1
./configure --prefix=/home/sharexu/software/valgrind/
make
make install
```

首先，需要下载安装包，

```
wget http://www.valgrind.org/downloads/valgrind-3.8.1.tar.bz2
```

解压安装包：

```
tar xvf valgrind-3.8.1.tar.bz2
```

进入文件夹：

```
cd valgrind-3.8.1
```

指定安装路径和生成 makefile：

```
./configure --prefix=/home/sharexu/software/valgrind/
```

从 makefile 中读取指令，然后编译：

```
make
```

从 makefile 中读取指令，安装到指定位置。

```
make install
```

5.5.5 Valgrind 使用

为了使 Valgrind 发现的错误更精确，如能够定位到源代码行，建议在编译时加上 `-g` 参数。

【例 5.9】访问非法内存。

```
#include<iostream>
#include<stdlib.h>
using namespace std;
void func(){
    int *x=(int *)malloc( 10 * sizeof ( int ) );
    x[10]=0;
}.
int main(){
    func();
    cout<<"done"<<endl;
    return 0;
}
```

用 `g++ -g -o test test.cpp` 命令编译之后，`./test` 执行后输出：`done`。提示成功。

接下来用 Valgrind 来分析下其中的内存使用。Valgrind 的参数分为两类，一类是 core 的参数，它对所有的工具都适用；另外一类就是具体某个工具如 Memcheck 的参数。Valgrind 默认的工具就是 Memcheck，也可以通过“`--tool=tool name`”指令指定其他的工具。Valgrind 提供了大量的参数满足用户特定的调试需求，具体可参考其用户手册。

这个例子将使用 Memcheck，于是可以输入命令如下：

```
/home/sharexu/software/valgrind/bin/valgrind ./test
```

执行结果如图 5-46 所示。

```
[sharexuglinux 0509]$ /home/sharexu/software/valgrind/bin/valgrind ./test
==27756== Memcheck, a memory error detector
==27756== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==27756== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==27756== Command: ./test
==27756==
==27756== Invalid write of size 4
==27756==   at 0x400072: func() (test.cpp:6)
==27756==   by 0x400082: main (test.cpp:9)
==27756== Address 0x5963060 is 0 bytes after a block of size 40 alloc'd
==27756==   at 0x4C278FE: malloc (vg_replace_malloc.c:270)
==27756==   by 0x400065: func() (test.cpp:5)
==27756==   by 0x400082: main (test.cpp:9)
==27756==
done
==27756==
==27756== HEAP SUMMARY:
==27756==   in use at exit: 40 bytes in 1 blocks
==27756==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==27756==
==27756== LEAK SUMMARY:
==27756==   definitely lost: 40 bytes in 1 blocks
==27756==   indirectly lost: 0 bytes in 0 blocks
==27756==   possibly lost: 0 bytes in 0 blocks
==27756==   still reachable: 0 bytes in 0 blocks
==27756==   suppressed: 0 bytes in 0 blocks
==27756== Rerun with --leak-check-full to see details of leaked memory
==27756==
==27756== For counts of detected and suppressed errors, rerun with: -v
==27756== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 0)
[sharexuglinux 0509]$
```

图 5-46 用 Valgrind 分析例 5.9 编译出来的目标文件

首先，图 5-46 左边显示类似行号的数字（27756）表示的是进程 id，下面是 Valgrind 的版本信息，如下所示：

```

==27756== Memcheck, a memory error detector
==27756== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==27756== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==27756== Command: ./test
==27756==

```

我们先把程序的行号都给显示出来，方便进一步分析，如图 5-47 所示。

（1）下面的提示了 Valgrind 通过运行被测试程序发现的一些内存问题。

1）这是一个对内存的非法写操作，非法写操作的内存是 4Byte。

2）发生错误时的函数堆栈，具体的源代码行号是第 6 行。

3）非法写操作的具体地址空间。

```

[sharex@linux 0509]$ vim test.cpp
1 #include <iostream>
2 #include <stdlib.h>
3 using namespace std;
4 void func(){
5     int *x=(int *)malloc( 10 * sizeof (int ) );
6     x[10]=1;
7 }
8 int main(){
9     func();
10    cout<< "done" << endl;
11    return 0;
12 }

```

图 5-47 例 5.9 程序按行显式

```

Invalid write of size 4
at 0x400872: func() (test.cpp:6)
by 0x400882: main (test.cpp:9)
Address 0x5963068 is 0 bytes after a block of size 40 alloc'd
at 0x4C278FE: malloc (vg_replace_malloc.c:270)
by 0x400865: func() (test.cpp:5)
by 0x400882: main (test.cpp:9)

```

下面是对发现的内存问题和内存泄漏问题的总结。程序结束时，程序泄漏了 40Byte 的内存，如下所示。

```

HEAP SUMMARY:
  in use at exit: 40 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 40 bytes allocated

LEAK SUMMARY:
  definitely lost: 40 bytes in 1 blocks
  indirectly lost: 0 bytes in 0 blocks
  possibly lost: 0 bytes in 0 blocks
  still reachable: 0 bytes in 0 blocks
  suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of leaked memory

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)

```

例 5.9 的程序有 2 个问题：① fun 函数中动态申请的堆内存没有释放；②对堆内存的访问越界。

这两个问题都被 Valgrind 给检测出来了。

下面来看下有哪些常见的内存错误使用情况，并看看如何用 Valgrind 将其检测出来。

1. 使用未初始化的内存

对于位于程序中不同段的变量，其初始值是不同的，全局变量和静态变量初始值为 0，而局部变量和动态申请的变量，其初始值为随机值。如果程序使用了为随机值的变量，那么程序的行为就变得不可预期。

下面的例 5.10 就是一种常见的使用了未初始化的变量的情况。数组 `a` 是局部变量，其初始值为随机值，而在初始化时并没有给其所有数组成员初始化，如此在接下来使用这个数组时就潜在有内存问题。

【例 5.10】使用未初始化的内存例子。

```
#include<iostream>
using namespace std;
int main(){
    int a[5];
    int i,s=0;
    a[0]=a[1]=a[3]=a[4]=0;
    for(i=0;i<5;i++){
        s=s+a[i];
    }
    if(s==33)
        cout<<"sum is 33"<<endl;
    else
        cout<<"sum is not 33"<<endl;
    return 0;
}
```

用 `g++ -g -o test test.cpp` 命令编译之后，`./test` 执行后输出：

```
sum is not 33
```

接下来使用 `memcheck` 模块来分析内存使用情况。执行以下命令：

```
/home/sharexu/software/valgrind/bin/valgrind --tool=memcheck ./test
```

结果如图 5-48 所示。

```
[sharexu@linux 0510]$ /home/sharexu/software/valgrind/bin/valgrind --tool=memcheck ./test
==27735== Memcheck, a memory error detector
==27735== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==27735== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==27735== Command: ./test
==27735==
==27735== Conditional jump or move depends on uninitialised value(s)
==27735== at 0x400864: main (test.cpp:9)
sum is not 33
==27735==
==27735== HEAP SUMMARY:
==27735== in use at exit: 0 bytes in 0 blocks
==27735== total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==27735==
==27735== All heap blocks were freed -- no leaks are possible
==27735==
==27735== For counts of detected and suppressed errors, rerun with: -v
==27735== Use --track-origins=yes to see where uninitialised values come from
==27735== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
[sharexu@linux 0510]$
```

图 5-48 用 `valgrind` 分析例 5.10 程序生成的目标文件

输出结果显示，在该程序第 11 行中，程序的跳转依赖于一个未初始化的变量 (`a[2]`)。`valgrind` 准确地发现了上述程序中存在的问题。

2. 内存读写越界

内存读写越界是指访问了没有权限访问的内存地址空间，比如访问数组时越界、对动态内存访问时超出了申请的内存大小范围。下面的程序例 5.11 就是一个典型的数组越界问题。pt 是一个局部数组变量，其大小为 4，p 初始指向 pt 数组的起始地址，但在对 p 循环叠加后，p 超出了 pt 数组的范围，如果此时再对 p 进行写操作，那么后果将不可预期。

【例 5.11】内存读写越界例子。

```
#include<stdlib.h>
#include<iostream>
using namespace std;
int main(){
    int len=4;
    int *pt=(int *)malloc(len*sizeof(int));
    int *p=pt;
    for(int i=0;i<len;i++){
        p++;
        *p=5;
        cout<<"the value of p is "<<*p<<endl;
        return 0;
    }
```

用 g++ -g -o test test.cpp 命令编译之后，./test 执行结果如下：

```
the value of p is 5
```

接下来我们使用 memcheck 模块来分析内存使用情况。执行以下命令：

```
/home/sharexu/software/valgrind/bin/valgrind --tool=memcheck ./test
```

执行结果如图 5-49 所示。

```
[sharexu@linux 0511]$ /home/sharexu/software/valgrind/bin/valgrind --tool=memcheck ./test
==27732== Memcheck, a memory error detector
==27732== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==27732== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==27732== Command: ./test
==27732==
==27732== Invalid write of size 4
==27732== at 0x4008f4: main (test.cpp:11)
==27732== Address 0x5963050 is 0 bytes after a block of size 16 alloc'd
==27732== at 0x4c278fe: malloc (vg_replace_malloc.c:270)
==27732== by 0x4008c4: main (test.cpp:6)
==27732==
==27732== Invalid read of size 4
==27732== at 0x4008fe: main (test.cpp:12)
==27732== Address 0x5963050 is 0 bytes after a block of size 16 alloc'd
==27732== at 0x4c278fe: malloc (vg_replace_malloc.c:270)
==27732== by 0x4008c4: main (test.cpp:6)
==27732==
the value of p is 5
==27732==
==27732== HEAP SUMMARY:
==27732== in use at exit: 16 bytes in 1 blocks
==27732== total heap usage: 1 allocs, 0 frees, 16 bytes allocated
==27732==
==27732== LEAK SUMMARY:
==27732== definitely lost: 16 bytes in 1 blocks
==27732== indirectly lost: 0 bytes in 0 blocks
==27732== possibly lost: 0 bytes in 0 blocks
==27732== still reachable: 0 bytes in 0 blocks
==27732== suppressed: 0 bytes in 0 blocks
==27732== Rerun with --leak-check=full to see details of leaked memory
==27732==
==27732== For counts of detected and suppressed errors, rerun with: -v
==27732== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 6 from 6)
[sharexu@linux 0511]$
```

图 5-49 用 valgrind 分析例 5.11 程序生成的目标文件

输出结果显示, 在该程序的第 11 行, 进行了非法的写操作; 在第 12 行, 进行了非法读操作。Valgrind 准确地发现了上述问题。

3. 内存覆盖

C 语言的强大和可怕之处在于其可以直接操作内存, C 标准库中提供了大量这样的函数, 比如 `strcpy`、`strncpy`、`memcpy`、`strcat` 等, 这些函数有一个共同的特点就是需要设置源地址 (src) 和目标地址 (dst), 且 src 和 dst 指向的地址不能发生重叠, 否则结果将不可预期。

下面就是一个 src 和 dst 发生重叠的例子。在 15 与 17 行中, src 和 dst 所指向的地址相差 20, 但指定的复制长度却是 21, 这样就会把之前的值覆盖。第 24 行程序类似, `src(x+20)` 与 `dst(x)` 所指向的地址相差 20, 但 dst 的长度却为 21, 这样也会发生内存覆盖。

【例 5.12】内存覆盖例子。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main(){
    char x[50];
    int i;
    for(i=0;i<50;i++){
        x[i]=i+1;
        strncpy(x+20,x,20);
        strncpy(x+20,x,21);
        strncpy(x,x+20,20);
        strncpy(x,x+20,21);
        x[39]='\0';
        strcpy(x,x+20);
        x[39]=39;
        x[40]='\0';
        strcpy(x,x+20);
    }
    return 0;
}
```

用 `g++ -g -o test test.cpp` 命令编译之后, `./test` 执行后发现没有任何输出。接下来使用 `memcheck` 模块来分析内存使用情况。执行以下命令: `/home/sharexu/software/valgrind/bin/valgrind --tool=memcheck ./test`

执行结果如图 5-50 所示。

输出结果显示上述程序中第 10、12、17 行, 源地址和目标地址设置出现重叠, Valgrind 均准确地发现了上述问题。

4. 动态内存管理错误

常见的内存分配方式分 3 种: 静态存储、栈上分配、堆上分配。全局变量属于静态存储, 它们是在编译时就被分配了存储空间; 函数内的局部变量属于栈上分配; 而最灵活的内存使用方式当属堆上分配, 也叫作内存动态分配。常用的内存动态分配函数包括: `malloc`、`alloc`、

realloc、new 等，动态释放函数包括 free 和 delete 等。

一旦成功申请了动态内存，就需要自己对其进行内存管理，而这又是最容易犯错误的。常见的内存动态管理错误包括以下几种。

```
[sharexu@linux 0512]$ /home/sharexu/software/valgrind/bin/valgrind --tool=memcheck ./test
==27726== Memcheck, a memory error detector
==27726== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==27726== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==27726== Command: ./test
==27726==
==27726== Source and destination overlap in strcpy(0x7ff000479, 0x7ff000465, 21)
==27726==   at 0x4C290FF: strcpy (mc_replace_strmem.c:472)
==27726==   by 0x40064A: main (test.cpp:10)
==27726==
==27726== Source and destination overlap in strcpy(0x7ff000465, 0x7ff000479, 21)
==27726==   at 0x4C290FF: strcpy (mc_replace_strmem.c:472)
==27726==   by 0x400682: main (test.cpp:12)
==27726==
==27726== Source and destination overlap in strcpy(0x7ff000450, 0x7ff000464)
==27726==   at 0x4C28F48: strcpy (mc_replace_strmem.c:438)
==27726==   by 0x4006BC: main (test.cpp:17)
==27726==
==27726==
==27726== HEAP SUMMARY:
==27726==   in use at exit: 0 bytes in 0 blocks
==27726== total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==27726==
==27726== All heap blocks were freed -- no leaks are possible
==27726==
==27726== For counts of detected and suppressed errors, rerun with: -v
==27726== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 6 from 6)
[sharexu@linux 0512]$
```

图 5-50 用 valgrind 分析例 5.12 程序生成的目标文件

(1) 申请和释放不一致。

由于 C++ 兼容 C，而 C 与 C++ 的内存申请和释放函数是不同的，因此在 C++ 程序中，就有两套动态内存管理函数。一条不变的规则就是采用 C 方式申请的内存就用 C 方式释放；用 C++ 方式申请的内存，用 C++ 方式释放。也就是用 malloc/alloc/realloc 方式申请的内存，用 free 释放；用 new 方式申请的内存用 delete 释放。但在上述程序第 11 行中，用 malloc 方式申请了内存却用 delete 来释放，虽然这在很多情况下不会有问题，但这绝对是潜在的问题。

(2) 申请和释放不匹配。

申请了多少内存，在使用完成后就要释放多少。如果没有释放，或者少释放了就是内存泄露；多释放了也会产生问题。上述程序中，指针 p 和 pt 指向的是同一块内存，却被先后释放两次。

(3) 释放后仍然读写。

本质上说，系统会在堆上维护一个动态内存链表，如果被释放，就意味着该块内存可以继续被分配给其他部分，如果内存被释放后再访问，就可能覆盖其他部分的信息，这是一种严重的错误，上述程序第 13 行中就在释放后仍然写这块内存。

下面的例 5.13，就包括了内存动态管理中常见的错误。

【例 5.13】动态内存管理错误例子。

```
#include<iostream>
#include<stdlib.h>
```

```

int main(){
    int i;
    char *p= (char *)malloc(10);
    char *pt=p;
    for(i=0;i<10;i++){
        p[i]='z';
    }
    delete p;
    pt[1]='x';
    free(pt);
    return 0;
}

```

用 `g++ -g -o test test.cpp` 命令编译之后, `./test` 执行结果如图 5-51 所示。

```

[sharexu@linux 0513]$ ./test
*** glibc detected *** ./test: double free or corruption (fasttop): 0x000000000e51010 ***
----- Backtrace: -----
/lib64/libc.so.6(0x75e66)[0x7fec824ce66]
./test[0x4007b9]
/lib64/libc.so.6(_libc_start_main+0xfdf)(0x7fec81f5d5d)
./test[0x400699]
----- Memory map: -----
00400000-00401000 r-xp 00000000 ca:01 262213 /home/sharexu/charpter05/0513/test
00600000-00601000 rw-p 00000000 ca:01 262213 /home/sharexu/charpter05/0513/test
00e51000-00e72000 rw-p 00000000 00:00 0 [heap]
7ffec81d7000-7ffec8361000 r-xp 00000000 ca:01 729100 /lib64/libc-2.12.so
7ffec8361000-7ffec8561000 --p 0018a000 ca:01 729100 /lib64/libc-2.12.so
7ffec8561000-7ffec8565000 r--p 0018a000 ca:01 729100 /lib64/libc-2.12.so
7ffec8565000-7ffec8566000 rw-p 0018e000 ca:01 729100 /lib64/libc-2.12.so
7ffec8566000-7ffec85ab000 rw-p 00000000 00:00 0
7ffec856b000-7ffec8591000 r-xp 00000000 ca:01 729090 /lib64/libgcc-s-4.4.6-20120305.so.1
7ffec8591000-7ffec8720000 --p 00016000 ca:01 729090 /lib64/libgcc-s-4.4.6-20120305.so.1
7ffec8720000-7ffec8721000 r-p 00015000 ca:01 729090 /lib64/libc-2.12.so
7ffec8721000-7ffec8804000 r-xp 00000000 ca:01 729347 /lib64/libc-2.12.so
7ffec8804000-7ffec8a03000 --p 00083000 ca:01 729347 /lib64/libc-2.12.so
7ffec8a03000-7ffec8a04000 r-p 00082000 ca:01 729347 /lib64/libc-2.12.so
7ffec8a04000-7ffec8a05000 r-p 00083000 ca:01 729347 /lib64/libc-2.12.so
7ffec8a05000-7ffec8aed000 r-xp 00000000 ca:01 672481 /usr/lib64/libstdc++.so.6.0.13
7ffec8aed000-7ffec8ced000 --p 000e0000 ca:01 672481 /usr/lib64/libstdc++.so.6.0.13
7ffec8ced000-7ffec8cf4000 r-p 000e0000 ca:01 672481 /usr/lib64/libstdc++.so.6.0.13
7ffec8cf4000-7ffec8cf6000 rw-p 000ef000 ca:01 672481 /usr/lib64/libstdc++.so.6.0.13
7ffec8cf6000-7ffec8db0000 rw-p 00000000 00:00 0
7ffec8db0000-7ffec8db2000 r-xp 00000000 ca:01 729317 /lib64/ld-2.12.so
7ffec8f1e000-7ffec8f23000 rw-p 00000000 00:00 0
7ffec8f2b000-7ffec8f2a000 rw-p 00000000 00:00 0
7ffec8f2a000-7ffec8f2b000 r-p 0001f000 ca:01 729317 /lib64/ld-2.12.so
7ffec8f2b000-7ffec8f2c000 rw-p 00020000 ca:01 729317 /lib64/ld-2.12.so
7ffec8f2c000-7ffec8f2d000 rw-p 00000000 00:00 0
7fffc437f000-7fffc4394000 rw-p 00000000 00:00 0 [stack]
7fffc43df000-7fffc43e0000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
Aborted (core dumped)
[sharexu@linux 0513]$

```

图 5-51 例 5.13 程序执行结果图

此时程序处于 `coredump` 状态, 接下来使用 `Memcheck` 模块来分析内存使用情况。执行以下命令: `/home/sharexu/software/valgrind/bin/valgrind --tool=memcheck ./test` 执行结果如图 5-52 所示。

图 5-52 显示, 第 11 行分配和释放函数不一致; 第 12 行发生非法写操作, 也就是往释放后的内存地址写值; 第 13 行释放内存函数无效。Valgrind 准确地发现了上述 3 个问题。

这时候可以顺便使用 `GDB` 来逐行看下究竟是哪一行导致程序处于 `coredump` 状态的。先在程序第 10、11、12、13、14 行设置断点, 如图 5-53 所示。

然后运行程序, 可见是在程序第 13 行时程序 `coredump` 了, `free(pt)` 语句释放了无效的

可是在调用完成之后，却没有相应的函数：nodefr 释放内存，这样内存中的这个树结构就无法被其他部分访问，造成了内存泄露。

在一个单独的函数中，每个人的内存泄露意识都是比较强的。但很多情况下，我们都会对 malloc/free 或 new/delete 做一些包装，以符合特定的需要，无法做到在一个函数中进行既使用又释放的操作。这个例子也说明了内存泄露最容易发生的地方：即两个部分的接口部分，一个函数申请内存，一个函数释放内存。并且这些函数由不同的人开发、使用，这样造成内存泄露的可能性就比较大了。这需要养成良好的单元测试习惯，将内存泄露问题消灭在初始阶段。

【例 5.13】内存泄漏例子。

tree.h 的代码是：

```
#ifndef _TREE_
#define _TREE_
typedef struct _node{
    struct _node *l;
    struct _node *r;
    char v;
}node;
node *mk(node *l, node *r, char val);
void nodefr(node *n);
#endif
```

tree.cpp 的代码是：

```
#include<stdlib.h>
#include"tree.h"
node *mk(node *l,node *r,char val){
    node *f=(node *)malloc(sizeof(*f));
    f->l=l;
    f->r=r;
    f->v=val;
    return f;
}
void nodefr(node *n){
    if(n){
        nodefr(n->l);
        nodefr(n->r);
        free(n);
    }
}
```

test.cpp 的代码是：

```
#include<iostream>
#include"tree.h"
int main(){
    node *tree1,*tree2,*tree3;
```



```

tree1=mk(mk(mk(0,0,'3'),0,'2'),0,'1');
tree2=mk(0,mk(0,mk(0,0,'6'),'5'),'4');
tree3=mk(mk(tree1,tree2,'8'),0,'7');
return 0;
}

```

makefile 的代码是：

```

test: test.o tree.o
    g++ -g -o test test.o tree.o
tree.o:tree.cpp tree.h
    g++ -g -c tree.cpp -o tree.o
test.o:test.cpp
    g++ -g -c test.cpp -o test.o

```

例 5.14 生成了一棵树，如图 5-55 所示。

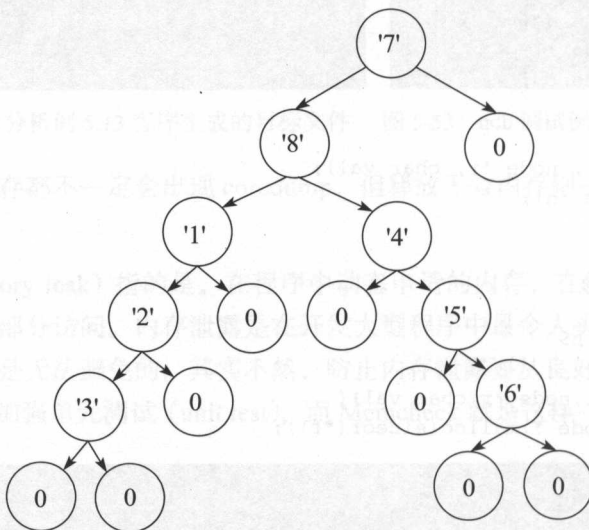


图 5-55 例 5.14 程序生成的树

接下来使用 Memcheck 模块来分析内存使用情况。执行以下命令：

```
/home/sharexu/software/valgrind/bin/valgrind --tool=memcheck ./test
```

执行结果如图 5-56 所示。

该示例程序是生成一棵树的过程，每个树节点的大小为 24（考虑内存对齐），共 8 个节点。从上述输出可以看出，所有的内存泄露都被发现。Memcheck 将内存泄露分为两种，一种是可能的内存泄露（possibly lost），另外一种是确定的内存泄露（definitely lost）。可能的内存泄露是指仍然存在某个指针能够访问某块内存，但该指针指向的已经不是该内存首地址。确定的内存泄露是指已经不能够访问这块内存。而确定的内存泄露又分为两种：

直接的 (direct) 和间接的 (indirect)。直接和间接的区别就是, 直接是没有任何指针指向该内存; 间接是指指向该内存的指针都位于内存泄露处。在上述的例子中, 根节点是直接泄露, 而其他节点是间接泄露。

```
[sharexu@linux 0513]$ /home/sharexu/software/valgrind/bin/valgrind --tool=memcheck ./test
==26825== Memcheck, a memory error detector
==26825== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==26825== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==26825== Command: ./test
==26825==
==26825== HEAP SUMMARY:
==26825==   in use at exit: 192 bytes in 8 blocks
==26825==   total heap usage: 8 allocs, 0 frees, 192 bytes allocated
==26825==
==26825== LEAK SUMMARY:
==26825==   definitely lost: 24 bytes in 1 blocks
==26825==   indirectly lost: 168 bytes in 7 blocks
==26825==   possibly lost: 0 bytes in 0 blocks
==26825==   still reachable: 0 bytes in 0 blocks
==26825==   suppressed: 0 bytes in 0 blocks
==26825== Rerun with --leak-check=full to see details of leaked memory
==26825==
==26825== For counts of detected and suppressed errors, rerun with: -v
==26825== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
[sharexu@linux 0513]$
```

图 5-56 用 valgrind 分析例 5.14 程序生成的目标文件

5.6 本章小结

本章讲述了各种调试程序的工具, 希望在读者实际编码过程中, 可助读者一臂之力。

前面都是讲一个程序里的各部分关系, 接下来要讲的是两台机器的程序间的关系。两台计算机之间, 需要通过网络协议才能进行通信。那么网络协议又是怎么样的呢? 这部分将在第 6 章为读者进行解答。

TCP 协议

信息交流的价值越来越凸显，那网络中程序之间如何通信，如每天打开浏览器浏览网页时，浏览器的程序怎么与 Web 服务器进行通信的？当使用 QQ 聊天时，QQ 怎么与服务器或你好友所在的 QQ 进行通信呢？这些都得靠网络通信，网络通信是通过网络将各个孤立的设备进行连接，通过信息交换实现人与人、人与计算机、计算机与计算机之间的通信。网络通信中最重要的就是网络通信协议，而网络通信协议中最重要的就是 TCP/IP 协议，本章将探索 TCP/IP 协议。

6.1 TCP 协议

6.1.1 网络模型

1. 七层网络模型

20 世纪 70 年代以来，国外一些主要计算机生产厂家先后推出了各自的网络体系结构，但它们都属于专用的。为使不同计算机厂家的计算机能够互相通信，以便在更大的范围内建立计算机网络，有必要建立一个国际范围的网络体系结构标准。为此，国际标准化组织 ISO 于 1981 年正式推荐了一个网络系统结构——七层参考模型，也叫作开放系统互连模型。由于这个标准模型的建立，使得各种计算机网络均向它靠拢，大大推动了网络通信的发展。这个 ISO 七层网络模型各层的名字、主要功能、对应的典型设备和传输单位如表 6-1 所示。

表 6-1 ISO 七层网络模型及其功能展示

层数	名字	主要功能	对应的典型设备	传输单位
7	应用层	提供应用程序间通信	计算机：应用程序，如 FTP、SMTP、HTTP 等	程序级数据
6	表示层	处理数据格式、数据加密等	计算机：编码方式，如图像编解码、URL 字段传输编码等	程序级数据
5	会话层	建立、维护和管理会话	计算机：建立会话，如 session 认证、断点续传	程序级数据
4	传输层	建立主机端到端连接	计算机：进程和端口	数据段 (segment)
3	网络层	寻址和路由选择	网络：路由器、防火墙、多层交换机	数据包 (packet)
2	数据链路层	提供介质访问、链路管理等	网络：网卡、网桥、交换机	帧 (frame)
1	物理层	比特流传输	网络：中继器、集线器、网线和 HUB	比特 (bit)

这个七层网络模型在数据的传输过程中还会对数据进行封装，封装过程如图 6-1 所示。

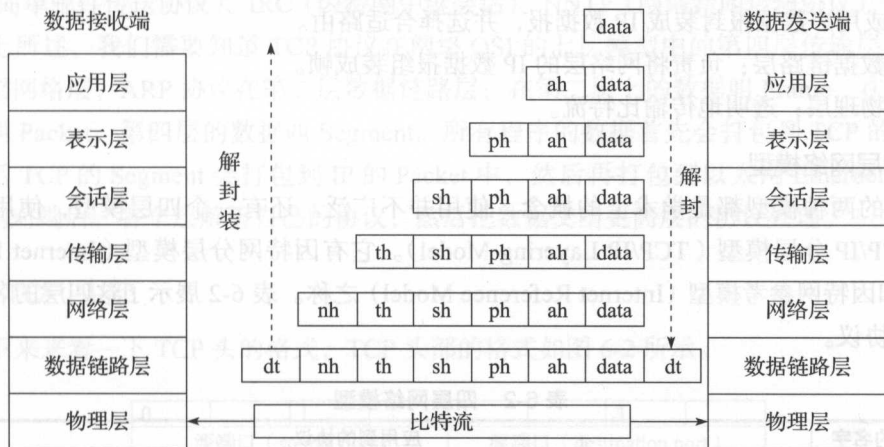


图 6-1 ISO 七层网络模型的数据封装与解封装

在 ISO 七层网络模型中，当一台主机需要传送用户的数据 (data) 时，数据首先通过应用层的接口进入应用层。在应用层，用户的数据被加上应用层的报头 (Ppplication Header, AH)，形成应用层协议数据单元 (Protocol Data Unit, PDU)，然后被递交到下层表示层。表示层并不“关心”上层应用层的数据格式而是把整个应用层递交的数据包看成是一个整体 (应用层数据) 进行封装，即加上表示层的报头 (Presentation Header, PH)。然后，递交到下层会话层。同样，会话层、传输层、网络层 (假设用 TCP 传输，则是 TCP 数据 + IP 包头)、数据链路层 (把上层的 TCP 数据 + IP 包头统一称为帧数据，即帧头 + 帧数据 + 帧尾 (CRC)) 也都要分别给上层递交下来的数据加上自己的报头。它们是：会话层报头 (Session Header, SH)、传输层报头 (Transport Header, TH)、网络层报头 (Network Header, NH) 和数据链路层报头 (Data link Header, DH)。其中，数据链路层还要给网络层递交的数据加上数据链路

层报尾 (Data link Termination, DT) 形成最终的一帧数据。

当一帧数据通过物理层传送到目标主机的物理层时，该主机的物理层把它递交到上层——数据链路层。数据链路层负责去掉数据帧的帧头部 DH 和尾部 DT（同时还进行数据校验）。如果数据没有出错，则递交到上层网络层。同样，网络层、传输层、会话层、表示层、应用层也要做类似的工作。最终，原始数据被递交到目标主机的具体应用程序中。

2. 五层网络模型

大学教科书中一般用了一个五层模型的网络体系，这五层的名字与功能分别如下所述。

- (1) 应用层：确定进程之间通信的性质以满足用户需求。应用层协议有很多，如支持万维网应用的 http 协议、支持电子邮件的 SMTP 协议、支持文件传送的 ftp 协议，等等。
- (2) 运输层：负责主机间不同进程的通信。这一层中的协议有面向连接的 TCP（传输控制协议）、无连接的 UDP（用户数据报协议）；数据传输的单位称为报文段或用户数据报。
- (3) 网络层：负责分组交换网中不同主机间的通信。作用为：发送数据时，将运输层中的报文段或用户数据报封装成 IP 数据报，并选择合适路由。
- (4) 数据链路层：负责将网络层的 IP 数据报组装成帧。
- (5) 物理层：透明地传输比特流。

3. 四层网络模型

前面的两种模型都是学术上的概念，使用并不广泛。还有一个四层模型，使用最为广泛——TCP/IP 分层模型 (TCP/IP Layering Model)。它有因特网分层模型 (Internet Layering Model) 和因特网参考模型 (Internet Reference Model) 之称。表 6-2 展示了这四层的名字和所应用到的协议。

表 6-2 四层网络模型

层数	层的名字	应用到的协议										
4	应用层	DNS	FINGER	WHOIS	FTP	HTTP	GOPHER	TELNET	IRC	SMTP	USERNET	其他
3	传输层	TCP					UDP					
2	网间层	TCMP			IP							
1	网络接口	ARP/RARP			其他							

TCP/IP 协议被组织成 4 个概念层，其中有 3 层对应于 ISO 参考模型中的相应层。ICP/IP 协议族并不包含物理层和数据链路层，因此它不能独立完成整个计算机网络系统的功能，必须与许多其他的协议协同工作。

TCP/IP 分层模型的 4 个协议层分别完成以下的功能。

(1) 网络接口层。

网络接口层包括用于协作 IP 数据在已有网络介质上传输的协议。实际上 TCP/IP 标准并不定义与 ISO 数据链路层和物理层相对应的功能。相反，它定义了像 APP (Address Resolution Protocol, 地址解析协议) 这样的协议，提供 TCP/IP 协议的数据结构和实际物理硬

件之间的接口。

(2) 网间层。

网间层对应于 OSI 七层参考模型的网络层。本层包含 IP 协议、RIP 协议（Routing Information Protocol，路由信息协议），负责数据的包装、寻址和路由。同时还包含 ICMP（Internet Control Message Protocol，网间控制报文协议）用来提供网络诊断信息。

(3) 传输层。

传输层对应于 OSI 七层参考模型的传输层，它提供两种端到端的通信服务。其中 TCP 协议（Transmission Control Protocol）提供可靠的数据流运输服务，UDP 协议（Use Datagram Protocol）提供不可靠的用户数据报服务。这其中的 TCP 协议是本章将要讨论的重点。

(4) 应用层。

应用层对应于 OSI 七层参考模型的应用层和表示层。因特网的应用层协议包括 Finger、Whois、FTP（文件传输协议）、Gopher、HTTP（超文本传输协议）、Telnet（远程终端协议）、SMTP（简单邮件传送协议）、IRC（因特网中继会话）、NNTP（网络新闻传输协议）等。

综上所述，我们需要知道 TCP 协议在网络 OSI 的七层模型中的第四层传输层，IP 协议在第三层网络层，ARP 协议在第二层数据链路层；在第二层上的数据叫 Frame，在第三层上的数据叫 Packet，第四层的数据叫 Segment。所有程序的数据首先会打包到 TCP 的 Segment 中，然后 TCP 的 Segment 会打包到 IP 的 Packet 中，然后再打包到以太网 Ethernet 的 Frame 中，传到对端后，各个层解析自己的协议，然后把数据交给更高层的协议处理。

6.1.2 TCP 头部

接下来来看一下 TCP 头的格式，TCP 头部的格式如图 6-2 所示。

0		1		2		3	
源端口（source port）				源端口（destination port）			
32位序号（sequence number）							
32位确认号（acknowledgment number）							
offset	reserved	标志位tcp flags CEUAPRSF		16位窗口大小（window size）			
16位检验和（checksum）				16位紧急指针（urgent pointer）			
tcp选项（tcp options）							

图 6-2 TCP 头部

TCP 头部里每一个字段都为管理 TCP 连接和控制数据流起了重要作用。

(1) 16 位端口号（port number）：告知主机该报文段是来自哪里（源端口）以及传给哪

个上层协议或应用程序（目的端口）的。进行 TCP 通信时，客户端通常使用系统自动选择的临时端口号，而服务器则使用知名服务端口号。所有知名服务使用的端口号都定义在 `/etc/services` 文件中。

(2) 32 位序号 (sequence number): 一次 TCP 通信（从 TCP 连接建立到断开）过程中某一个传输方向上的字节流的每个字节的编号。假设主机 A 和主机 B 进行 TCP 通信，A 发送给 B 的第一个 TCP 报文段中，序号值被系统初始化为某个随机值 ISN (Initial Sequence Number, 初始序号值)。那么在该传输方向上（从 A 到 B），后续的 TCP 报文段中序号值将被系统设置成 ISN 加上该报文段所携带数据的第一个字节在整个字节流中的偏移。例如，某个 TCP 报文段传送的数据是字节流中的第 1025 ~ 2048 字节，那么该报文段的序号值就是 ISN+1025。另外一个传输方向（从 B 到 A）的 TCP 报文段的序号值也具有相同的含义。

(3) 32 位确认号 (acknowledgement number): 用作对另一方发送来的 TCP 报文段的响应。其值是收到的 TCP 报文段的序号值加 1。假设主机 A 和主机 B 进行 TCP 通信，那么 A 发送出的 TCP 报文段不仅携带自己的序号，而且包含对 B 发送来的 TCP 报文段的确认号。反之，B 发送出的 TCP 报文段也同时携带自己的序号和对 A 发送来的报文段的确认号。

(4) 4 位头部长度 (header length): 标识该 TCP 头部有多少个 32bit (4 Byte)。因为 4 位最大能表示 15，所以 TCP 头部最长是 60 Byte。

(5) 6 位标志位包含如下几项。

1) URG 标志，表示紧急指针 (urgent pointer) 是否有效。

2) ACK 标志，表示确认号是否有效，一般称携带 ACK 标志的 TCP 报文段为“确认报文段”。

3) PSH 标志，提示接收端应用程序应该立即从 TCP 接收缓冲区中读走数据，为接收后续数据腾出空间（如果应用程序不将接收到的数据读走，它们就会一直停留在 TCP 接收缓冲区中）。

4) RST 标志，表示要求对方重新建立连接，一般称携带 RST 标志的 TCP 报文段为“复位报文段”。

5) SYN 标志，表示请求建立一个连接，一般称携带 SYN 标志的 TCP 报文段为“同步报文段”。

6) FIN 标志，表示通知对方本端要关闭连接了，一般称携带 FIN 标志的 TCP 报文段为“结束报文段”。

(6) 16 位窗口大小 (window size): 是 TCP 流量控制的一个手段。这里说的窗口，指的是接收通告窗口 (Receiver Window, RWND)。它告诉对方本端的 TCP 接收缓冲区还能容纳多少字节的数据，这样对方就可以控制发送数据的速度。

(7) 16 位校验和 (TCP checksum): 由发送端填充，接收端对 TCP 报文段执行 CRC 算法，以检验 TCP 报文段在传输过程中是否损坏。注意，这个校验不仅包括 TCP 头部，也包括数据部分。这也是 TCP 可靠传输的一个重要保障。

(8) 16 位紧急指针 (urgent pointer): 是一个正的偏移量。它和序号字段的值相加表示最后一个紧急数据的下一字节的序号。因此, 确切地说, 这个字段是紧急指针相对当前序号的偏移, 不妨称之为“紧急偏移”。TCP 的紧急指针是发送端向接收端发送紧急数据的方法。我们将在后面讨论 TCP 紧急数据。

综上, 你需要注意如下几点。

- 1) TCP 的包是没有 IP 地址的, 那是 IP 层上的事, 但是有源端口和目的端口。
- 2) 一个 TCP 连接需要 4 个元组 (src_ip、src_port、dst_ip、dst_port) 来表示是同一个连接。准确说是五元组, 还有一个是协议。但因为这里只是强调 TCP 协议, 所以, 只说四元组。
- 3) Sequence Number 是包的序号, 用来解决网络包乱序 (reordering) 问题。
- 4) Acknowledgement Number 就是 ACK, 用于确认收到, 用来解决不丢包的问题。
- 5) Window 又叫 Advertised-Window, 也就是著名的滑动窗口 (Sliding Window), 用于解决流控问题。
- 6) TCP Flag, 也就是包的类型, 主要是用于操控 TCP 的状态机的。

6.1.3 TCP 状态流转

其实, 网络上的传输是没有连接的, TCP 也是一样的。而 TCP 所谓的“连接”, 其实只不过是通信的双方维护一个“连接状态”, 让它看上去好像有连接一样。所以, TCP 的状态变换是非常重要的。先来看一下著名的 3 次握手与 4 次挥手图, 如图 6-3 所示。

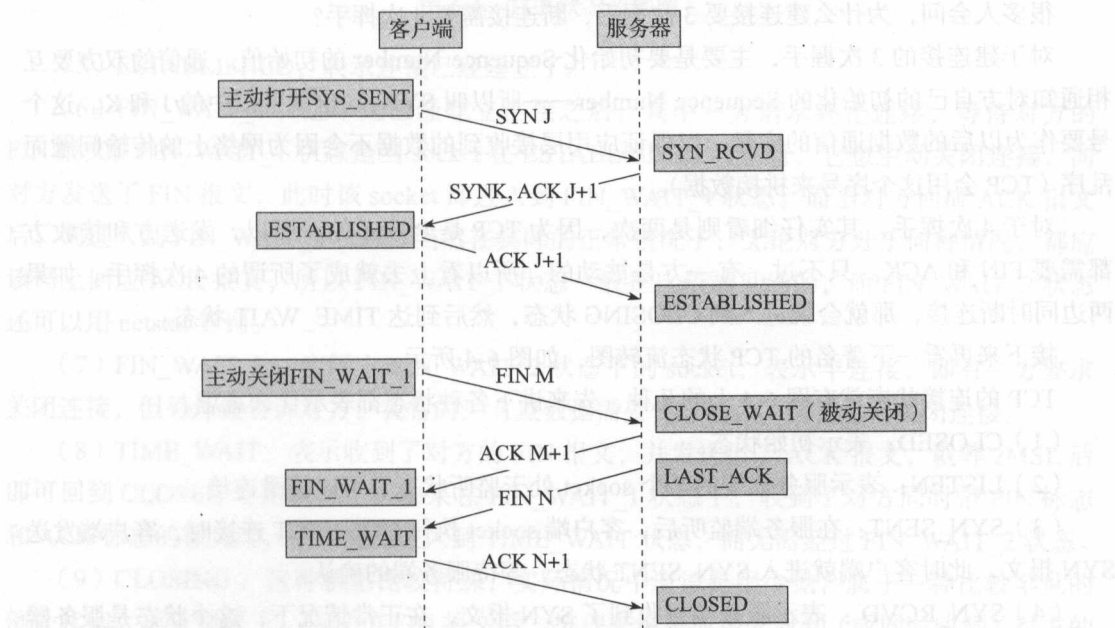


图 6-3 3 次握手与 4 次挥手

TCP 连接的建立可以简单地称为 3 次握手，而连接的中止则可以称为 4 次挥手。

(1) 建立连接。

在 TCP/IP 协议中，TCP 协议提供可靠的连接服务，采用三次握手建立一个连接。

1) 第一次握手：建立连接时，客户端发送 SYN 包 ($\text{SYN}=\text{J}$) 到服务器，并进入 SYN_SEND 状态，等待服务器确认。

2) 第二次握手：服务器收到 SYN 包，必须确认客户的 SYN ($\text{ACK}=\text{J}+1$)，同时自己也发送一个 SYN 包 ($\text{SYN}=\text{K}$)，即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态。

3) 第三次握手：客户端收到服务器的 SYN + ACK 包，向服务器发送确认包 ACK ($\text{ACK}=\text{K}+1$)，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成 3 次握手。

完成三次握手，客户端与服务器开始传送数据，也就是 ESTABLISHED 状态。

(2) 结束连接。

TCP 有一个特别的概念叫作半关闭，这个概念是说，TCP 的连接是全双工（可以同时发送和接收）连接，因此在关闭连接的时候，必须关闭传和送两个方向上的连接。客户机给服务器一个 FIN 的 TCP 报文，然后服务器返回给客户端一个确认 ACK 报文，并且发送一个 FIN 报文，当客户机回复 ACK 报文后（4 次握手），连接就结束了。

在建立连接的时候，通信的双方要互相确认对方的最大报文长度（MSS），以便通信。一般这个 SYN 长度是 MTU 减去固定 IP 首部和 TCP 首部长度。对于一个以太网，一般可以达到 1460 Byte。当然如果对于非本地的 IP，这个 MSS 可能就只有 536 Byte，而且，如果中间的传输网络的 MSS 更加的小的话，这个值还会变得更小。

很多人会问，为什么建连接要 3 次握手，断连接需要 4 次挥手？

对于建连接的 3 次握手，主要是要初始化 Sequence Number 的初始值。通信的双方要互相通知对方自己的初始化的 Sequence Number——所以叫 SYN，也就上图中的 J 和 K。这个号要作为以后的数据通信的序号，以保证应用层接收到的数据不会因为网络上的传输问题而乱序（TCP 会用这个序号来拼接数据）。

对于 4 次挥手，其实仔细看则是两次，因为 TCP 是全双工的，所以，发送方和接收方都需要 FIN 和 ACK。只不过，有一方是被动的，所以看上去就成了所谓的 4 次挥手。如果两边同时断连接，那就会进入到 CLOSING 状态，然后到达 TIME_WAIT 状态。

接下来再看一下著名的 TCP 状态流转图，如图 6-4 所示。

TCP 的连接状态就有图 6-4 上的几种，先来讲下各种状态都表示什么意思。

(1) CLOSED：表示初始状态。

(2) LISTEN：表示服务器端的某个 socket 处于监听状态，可以接受连接。

(3) SYN_SENT：在服务端监听后，客户端 socket 执行 CONNECT 连接时，客户端发送 SYN 报文，此时客户端就进入 SYN_SENT 状态，等待服务端的确认。

(4) SYN_RCVD：表示服务端接收到了 SYN 报文，在正常情况下，这个状态是服务器端的 socket 在建立 TCP 连接时的 3 次握手会话过程中的一个中间状态，很短暂，基本上用网

络查询工具 `netstat` 是很难看到这种状态的，除非特意写了一个客户端测试程序，故意将 3 次 TCP 握手过程中最后一个 ACK 报文不予发送。因此这种状态时，当收到客户端的 ACK 报文后，它会进入到 `ESTABLISHED` 状态。

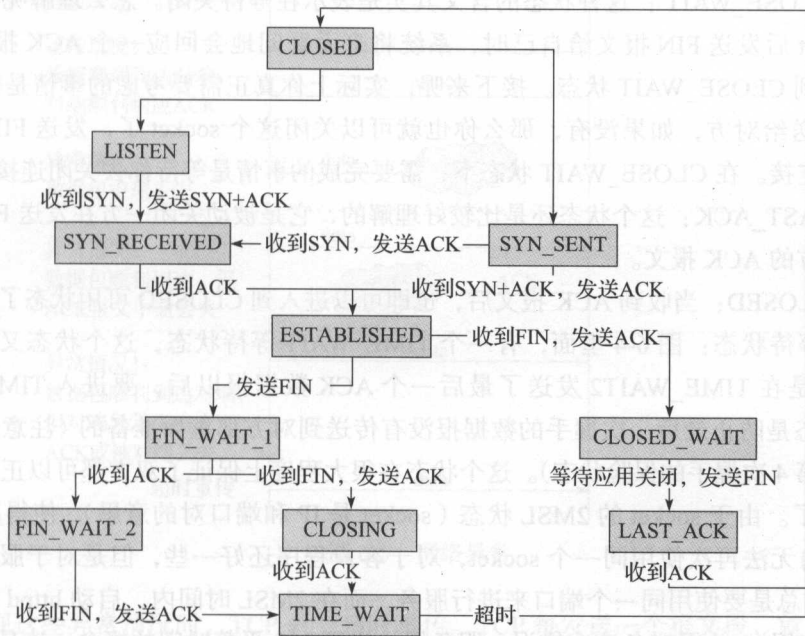


图 6-4 TCP 状态流转图

(5) `ESTABLISHED`：表示连接已经建立了。

(6) `FIN_WAIT_1`：这个是已经建立连接之后，其中一方请求终止连接，等待对方的 FIN 报文。`FIN_WAIT_1` 状态是当 socket 在 `ESTABLISHED` 状态时，它想主动关闭连接，向对方发送了 FIN 报文，此时该 socket 即进入到 `FIN_WAIT_1` 状态。而当对方回应 ACK 报文后，则进入到 `FIN_WAIT_2` 状态，当然在实际的正常情况下，无论对方处于何种情况，都应该马上回应 ACK 报文，所以 `FIN_WAIT_1` 状态一般是比较难见到的，而 `FIN_WAIT_2` 状态还可以用 `netstat` 看到。

(7) `FIN_WAIT_2`：实际上 `FIN_WAIT_2` 状态下的 socket，表示半连接，即有一方要求关闭连接，但另外还告诉对方：我暂时还有点数据需要传送给你，请稍后再关闭连接。

(8) `TIME_WAIT`：表示收到了对方的 FIN 报文，并发送出了 ACK 报文，就等 2MSL 后即可回到 `CLOSED` 可用状态了。如果在 `FIN_WAIT_1` 状态下，收到了对方同时带 FIN 标志和 ACK 标志的报文时，可以直接进入到 `TIME_WAIT` 状态，而无需经过 `FIN_WAIT_2` 状态。

(9) `CLOSING`：这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当发送 FIN 报文后，按理来说是应该先收到（或同时收到）对方的 ACK 报文，再收到对方的 FIN 报文。但是 `CLOSING` 状态表示你发送 FIN 报文后，并没有

收到对方的 ACK 报文，反而收到了对方的 FIN 报文。为什么会出现此种情况呢？其实细想一下，也不难得出结论：那就是如果双方几乎在同时关闭一个 socket 的话，那么就出现了双方同时发送 FIN 报文的情况，就会出现 CLOSING 状态，表示双方都正在关闭 socket 连接。

(10) CLOSE_WAIT：这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方关闭一个 socket 后发送 FIN 报文给自己时，系统将毫无疑问地会回应一个 ACK 报文给对方，此时则进入到 CLOSE_WAIT 状态。接下来呢，实际上你真正需要考虑的事情是察看你是否还有数据发送给对方，如果没有，那么你也就可以关闭这个 socket 了，发送 FIN 报文给对方，即关闭连接。在 CLOSE_WAIT 状态下，需要完成的事情是等待你去关闭连接。

(11) LAST_ACK：这个状态还是比较好理解的，它是被动关闭一方在发送 FIN 报文后，最后等待对方的 ACK 报文。

(12) CLOSED：当收到 ACK 报文后，也即可以进入到 CLOSED 可用状态了。

2MSL 等待状态：图 6-4 里面，有一个 TIME_WAIT 等待状态，这个状态又叫作 2MSL 状态，说的是在 TIME_WAIT2 发送了最后一个 ACK 数据报以后，要进入 TIME_WAIT 状态，这个状态是防止最后一次握手的数据报没有传送到对方那里而准备的（注意这不是 4 次握手，这是第 4 次握手的保险状态）。这个状态在很大程度上保证了双方都可以正常结束，但是问题也来了。由于 socket 的 2MSL 状态（socket 是 IP 和端口对的意思），使得应用程序在 2MSL 时间内无法再次使用同一个 socket，对于客户程序还好一些，但是对于服务程序（例如 httpd），它总是要使用同一个端口来进行服务，而在 2MSL 时间内，启动 httpd 就会出现错误（插口被使用）。为了避免这个错误，服务器给出了一个平静时间的概念，这是说在 2MSL 时间内，虽然可以重新启动服务器，但是这个服务器还是要平静地等待 2MSL 的时间才能进行下一次连接。

FIN_WAIT_2 状态：这就是著名的半关闭状态了，这是在关闭连接时，客户端和服务端两次握手之后的状态。在这个状态下，应用程序还有接收数据的能力，但是已经无法发送数据，但是也有一种可能是，客户端一直处于 FIN_WAIT_2 状态，而服务器则一直处于 WAIT_CLOSE 状态，直到应用层来决定关闭这个状态。

RST，同时打开和同时关闭：RST 是另一种关闭连接的方式，应用程序应该可以判断 RST 包的真实性，即是否为异常中止。而同时打开和同时关闭则是两种特殊的 TCP 状态，发生的概率很小。

6.1.4 TCP 超时重传

本节将讨论异常网络状况下（开始出现超时或丢包），TCP 如何控制数据传输以保证其承诺的可靠服务。

假设有这样的几种情况：

- 1) 数据顺利到底对端，对端顺利响应 ACK；
- 2) 数据包中途丢失；

3) 数据包顺利到达, 但 ACK 报文中途丢失;

4) 数据包梳理到达对端, 但对端异常未响应 ACK 或被对端丢弃。

这几种正常和异常的情况如图 6-5 所示。

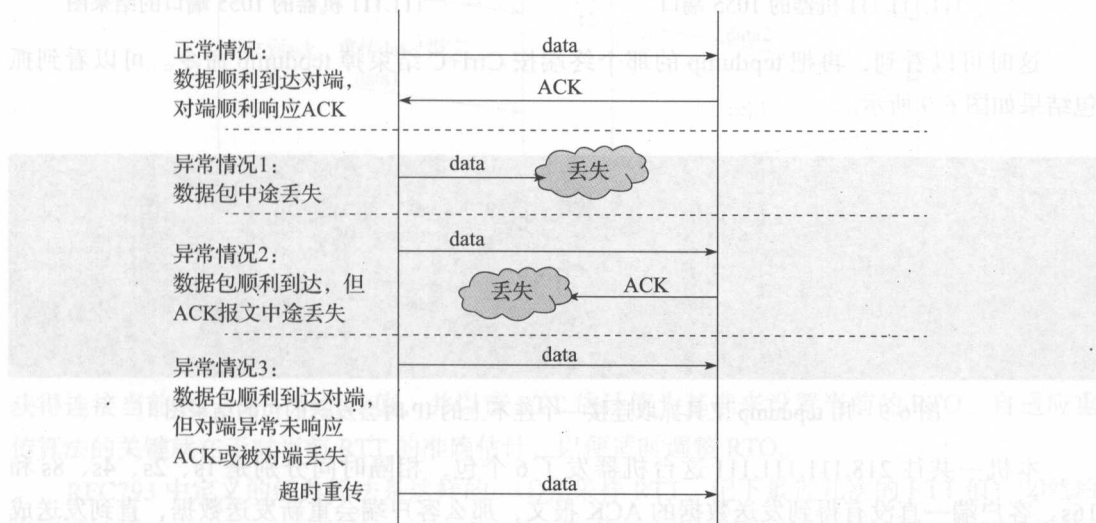


图 6-5 网络异常

当出现这些异常情况时, TCP 就会超时重传。TCP 每发送一个报文段, 就对这个报文段设置一次计时器。只要计时器设置的重传时间到了, 但还没有收到确认, 就要重传这一报文段, 这个就叫作“超时重传”。

这里用 telnet 和 tcpdump 工具来演示下 TCP 是怎么超时重传的。其中, telnet ip port 是查看某一个机器上的某一个端口是否可以访问。tcpdump-ieth1 'port 1055' 是用来抓取网卡 eth1 上的 1055 端口上的包, 如图 6-6 所示。

```
[root@linux ~]# tcpdump -ieth1 'port 1055'
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on eth1, link-type EN10MB (Ethernet), capture size 65535
bytes
```

图 6-6 用 tcpdump 工具抓 1055 端口上的包

我们先在一个终端里执行 tcpdump-ieth1 'port 1055' 命令, 看本机器是否使用 1055 端口来传送数据。

等了一会, 没有其他请求干扰后, 就可以再开一个终端, 并执行 telnet 218.111.111.111 1055 命令, 探测 218.111.111.111 1055 是否可以连得通, 如图 6-7 所示。

过了一会, 可以看到如图 6-8 所示的界面。

说明连接到 218.111.111.111 1055 失败了。


```
[sharexu@linux chapter06]$ telnet 218.111.111.111 1055
Trying 218.111.111.111...
```

图 6-7 用 telnet 命令尝试连接 IP 为 218.111.111.111 机器的 1055 端口

```
[sharexu@linux chapter06]$ telnet 218.111.111.111 1055
Trying 218.111.111.111...
telnet: connect to address 218.111.111.111: Connection timed out
[sharexu@linux chapter06]$
```

图 6-8 用 telnet 命令连接 IP 为 218.111.111.111 机器的 1055 端口的结果图

这时可以看到，再把 tcpdump 的那个终端按 Ctrl+C 结束掉 tcpdump 命令。可以看到抓包结果如图 6-9 所示。

```
[root@linux ~]# tcpdump -ieth1 'port 1055'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 65535 bytes
10:11:23.067137 IP 42.96.142.129.opsec-omi > 218.111.111.111.ansyslmd: Flags [S], seq 242514080, win 14600, options [mss 1460], length 0
10:11:24.066947 IP 42.96.142.129.opsec-omi > 218.111.111.111.ansyslmd: Flags [S], seq 242514080, win 14600, options [mss 1460], length 0
10:11:26.066966 IP 42.96.142.129.opsec-omi > 218.111.111.111.ansyslmd: Flags [S], seq 242514080, win 14600, options [mss 1460], length 0
10:11:30.066942 IP 42.96.142.129.opsec-omi > 218.111.111.111.ansyslmd: Flags [S], seq 242514080, win 14600, options [mss 1460], length 0
10:11:38.066951 IP 42.96.142.129.opsec-omi > 218.111.111.111.ansyslmd: Flags [S], seq 242514080, win 14600, options [mss 1460], length 0
10:11:54.066961 IP 42.96.142.129.opsec-omi > 218.111.111.111.ansyslmd: Flags [S], seq 242514080, win 14600, options [mss 1460], length 0
^C
6 packets captured
10 packets received by filter
0 packets dropped by kernel
[root@linux ~]#
```

图 6-9 用 tcpdump 工具抓取连接一个连不上的 IP 时会发送的包的结果图

本机一共往 218.111.111.111 这台机器发了 6 个包，相隔时间分别是 1s、2s、4s、8s 和 16s。客户端一直没有得到发送数据的 ACK 报文，那么客户端会重新发送数据，直到发送成功位置。那这个超时重传时间一般设置为多少才合适呢？影响超时重传机制协议效率的一个关键参数是 RTO（Retransmission TimeOut，重传超时时间）。RTO 指发送端发送数据后、重传数据前等待接受方收到该数据报文的 ack 时间。

RTO 的设置对于重传非常重要：

- 1) 设长了，重发就慢，没有效率，性能差；
- 2) 设短了，重发得就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

如果底层网络的传输特性是可预知的，那么重传机制的设计相对简单得多，可根据底层网络的传输时延的特性选择一个合适的 RTO，使协议的性能得到优化。但是 TCP 的底层网络环境是一个完全异构的互联结构。在实现端到端的通信时，不同端点之间传输通路的性能可能存在着巨大的差异，而且同一个 TCP 连接在不同的时间段上，也会由于不同的网络状态具有不同的传输时延。

因此，TCP 协议必须适应两个方面的时延差异：一个是达到不同目的端的时延的差异；另一个是统一连接上的传输时延随业务量负载的变化而出现的差异。为了处理这种底层网络传输特性的差异性和变化性，TCP 的重传机制相对于其他协议显然也更为复杂，其复杂性主要表现在对超时时间间隔的处理上。为此，TCP 协议使用自适应算法（Adaptive Retransmission Algorithm）以适应互联网分组传输时延的变化。这种算法的基本要点是 TCP 监视每个连接的性能（即传输时延），由每一个 TCP 的连接情况推算出合适的 RTO 值，当连接时延性能变化时，TCP 也能够相应地自动修改 RTO 的设定，以适应这种网络的变化。

为了动态地设置，TCP 引入了 RTT（Round Trip Time），也就是连接往返时间，指发送端从

发送 TCP 包开始到接收它的立即响应所耗费的传输时间。RTO 与 RTT 不同之处如图 6-10 所示。

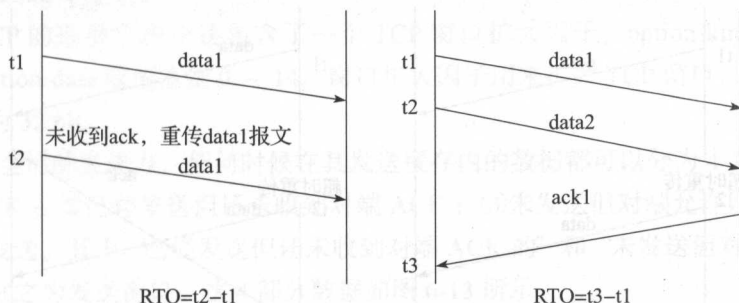


图 6-10 RTO 与 RTT 比较示意图

对一个连接而言，若能够了解端点间的 RTT（传输往返时间），则可根据 RTT 来设置合适的 RTO。显然，在任何时刻连接的 RTT 都是随机的，无法事先预知。TCP 通过测量来获得连接当前 RTT 的一个估计值，并以该 RTT 估计值为基准来设置当前的 RTO。自适应重传算法的关键就在于对当前 RTT 的准确估计，以便适时调整 RTO。

RFC793 中定义的经典算法是这样的：①先采样 RTT，记下最近几次的 RTT 值；②然后做平滑计算 SRTT(Smoothed RTT)。公式中的 α 取值为 0.8 ~ 0.9，这个算法叫加权移动平均，公式如下：

$$SRTT = \alpha * SRTT + (1 - \alpha) * RTT$$

接着开始计算 RTO，公式如下：

$$RTO = \min [UBOUND, \max [LBOUND, (\beta * SRTT)]]$$

式中：

UBOUND 是最大的 timeout 时间，上限值；LBOUND 是最小的 timeout 时间，下限值； β 值一般在 1.3 ~ 2.0。

但是上面的这个算法在重传的时候会出有一个终极问题——如果是用第一次的时间和 ACK 回来的时间做 RTT 样本，或是用重传的时间和 ACK 的时间做 RTT 样本，都会产生问题，这个被称为重传的多义性问题，如图 6-11 所示。

图 6-11 中，情况图 6-11a 是 ACK 没回来，所发重传，如果你计算第一次发送和 ACK 的时间，那么，明显算大了。情况图 6-11b 是 ACK 回来慢了，重传不一会，之前 ACK 就回来了。如果你是算重传的时间和 ACK 回来的时间，就会短了。

1987 年的时候出现了 Karn/Partridge Algorithm，这个算法的最大特点是——忽略重传，不把重传的 RTT 做采样。但是这样一来，又会引发一个大 Bug——如果在某一时间，网络出现抖动，突然变慢了，产生了比较大的延时，这个延时导致要重传所有的包（因为之前的 RTO 很小），这会导致 RTO 不会被更新，这是一个灾难。于是 Karn 算法用了一个取巧的方

式——只要一发生重传，就对现有的 RTO 值翻倍（这就是所谓的 Exponential backoff）。

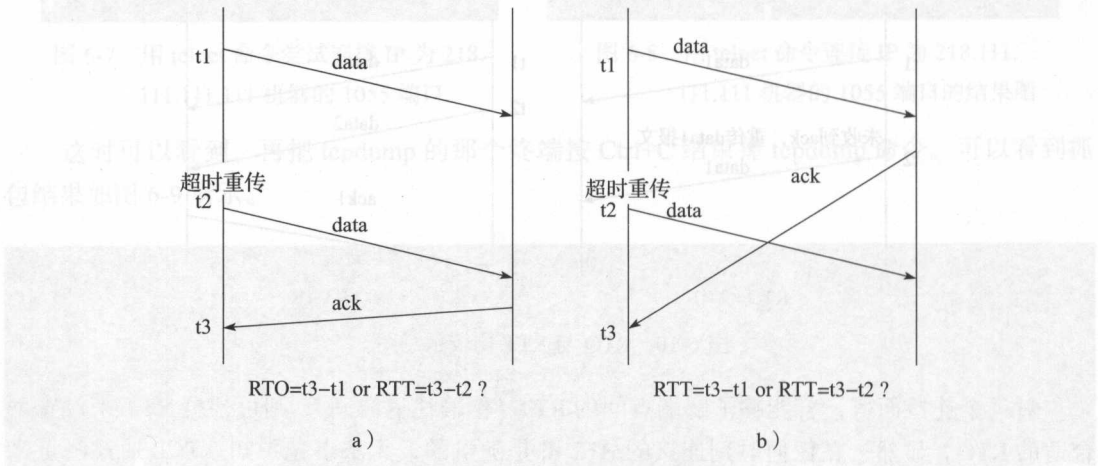


图 6-11 重传的多义性问题

Karm 算法的计算方式为：报文段每重传一次，就将重传时间增大一些，公式是：

新的重传时间 = $\gamma \times (\text{旧的重传时间})$

式中：系数 γ 的典型值是 2。

当不再发生报文段的重传时，才根据报文段的往返时延更新平均往返时延 RTT 和重传时间的数值。所以上面的例子，重传的时间为 1s、2s、4s、8s 和 16s，符合了 Karm 算法。

6.1.5 TCP 滑动窗口

TCP 的滑动窗口主要有两个作用：一是提供 TCP 的可靠性；二是提供 TCP 的流控特性。同时滑动窗口机制还体现了 TCP 面向字节流的设计思路。TCP 段中窗口的相关字段如图 6-12 所示。

0	1	2	3
源端口（source port）		目的端口（destination port）	
32位序号（sequence number）			
32位确认号（acknowledgment number）			
offset	reserved	标志位（tcp flags） CEUAPRSF	16位窗口大小（window size）
16位检验和（checksum）		16位紧急指针（urgent pointer）	
TCP选项（TCP options）			

图 6-12 TCP 头部中滑动窗口所处的位置

TCP 的窗口是一个 16bit 位字段，它代表的是窗口的字节容量，也就是 TCP 的标准窗口最大为 $2^{16}-1=65535$ 个字节。

另外在 TCP 的选项字段中还包含了一个 TCP 窗口扩大因子，option-kind 为 3，option-length 为 3，option-data 取值范围 0 ~ 14。窗口扩大因子用来扩大 TCP 窗口，可把原来 16bit 的窗口，扩大为 32 bit。

对于 TCP 会话的发送方，任何时候在其发送缓存内的数据都可以分为 4 类：①已经发送并得到对端 ACK；②已经发送但还未收到对端 ACK；③未发送但对端允许发送；④未发送且对端不允许发送。其中“已经发送但还未收到对端 ACK 的”和“未发送但对端允许发送的”这两部分数据称之为发送窗口，这 4 部分数据如图 6-13 所示。

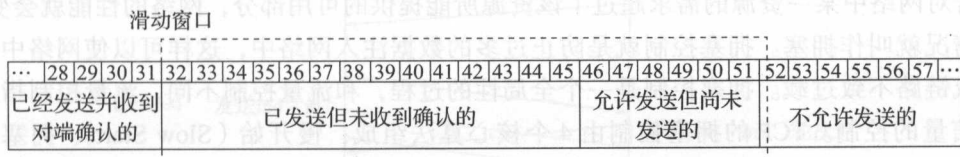


图 6-13 滑动窗口

当收到接收方新的 ACK 对于发送窗口中后续字节的确认时，窗口滑动原理如图 6-14 所示。

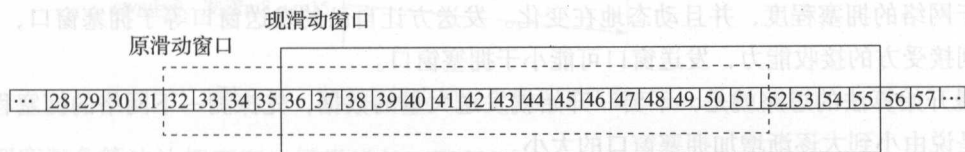


图 6-14 滑动窗口滑动示意图

对于 TCP 的接收方，在某一时刻在它的接收缓存内存在 3 种状态：①已接收；②未接收准备接收；③未接收并未准备接收（由于 ACK 直接由 TCP 协议栈回复，默认无应用延迟，不存在“已接收未回复 ACK”）。其中“未接收准备接收”称之为接收窗口。

TCP 是双工的协议，会话的双方都可以同时接收、发送数据。TCP 会话的双方都各自维护一个“发送窗口”和一个“接收窗口”。其中各自的“接收窗口”大小取决于应用、系统、硬件的限制（TCP 传输速率不能大于应用的数据处理速率）。各自的“发送窗口”则要求取决于对端通告的“接收窗口”，要求相同。

滑动窗口实现面向流的可靠性来源于“确认重传”机制。TCP 的滑动窗口的可靠性也是建立在“确认重传”基础上的。发送窗口只有收到对端对于本段发送窗口内字节的 ACK 确认，才会移动发送窗口的左边界。接收窗口只有在前面所有的段都确认的情况下才会移动左边界；在前面还有字节未接收但收到后面字节的情况下，窗口不会移动，并不对后续字节确认。以此确保对端会对这些数据重传。

TCP 的滑动窗口是动态的，我们可以想象成小学常见的一个数学题，一个水池，体积

V ，每小时进水量 V_1 ，出水量 V_2 ；当水池满了就不允许再注入了。如果有个液压系统控制水池大小，那么就可以控制水的注入速率和量；这样的水池就类似 TCP 的窗口。应用根据自身的处理能力变化，通过本端 TCP 接收窗口大小控制来对端的发送窗口进行流量限制。

应用程序在需要（如内存不足）时，通过 API 通知 TCP 协议栈缩小 TCP 的接收窗口。然后 TCP 协议栈在下个时间段发送时包含新的窗口大小通知给对端，对端按通知的窗口来改变发送窗口，以此达到减缓发送速率的目的。

6.1.6 TCP 拥塞控制

计算机网络中的带宽、交换结点中的缓存和处理机等，都是网络的资源。在某段时间内，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就会变坏。这种情况就叫作拥塞。拥塞控制就是防止过多的数据注入网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制是一个全局性的过程，和流量控制不同，流量控制指点对点通信量的控制。TCP 的拥塞控制由 4 个核心算法组成：慢开始（Slow Start）、拥塞避免（Congestion Avoidance）、快速重传（Fast Retransmit）和快速恢复（Fast Recovery）。

1. 慢开始和拥塞避免

发送方维持一个叫作拥塞窗口 $cwnd$ （congestion window）的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口，另外考虑到接受方的接收能力，发送窗口可能小于拥塞窗口。

慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。

慢开始的原理如下所述。

（1）当主机开始发送数据时，如果立即将较大的发送窗口的全部数据字节都注入到网络中，那么由于不清楚网络的情况，有可能引其网络拥塞。

（2）比较好的方法是试探一下，即从小到大逐渐增大发送端的拥塞控制窗口数值。

（3）在刚刚开始发送报文段时可先将拥塞窗口 $cwnd$ 设置为一个最大报文段的 MSS 的数值。在每收到一个对新报文段确认后，将拥塞窗口增加至多一个 MSS 的数值，当 $cwnd$ 足够大的时候，为了防止拥塞窗口 $cwnd$ 的增长引起网络拥塞，还需要另外一个变量，即慢开始门限 $ssthresh$ 。

这里用报文段的个数的拥塞窗口大小举例说明慢开始算法，实时拥塞窗口大小是以字节为单位的，如图 6-15 所示。

当然收到单个确认但此确认多个数据报的时候就加相应的数值。所以一次传输轮次之后拥塞窗口就加倍。这就是指数增长（ $y = 2^x$ ），和后面的拥塞避免算法的加法增长比较。为了防止 $cwnd$ 增长过大引起网络拥塞，还需设置一个慢开始门限 $ssthresh$ 状态变量。 $ssthresh$ 的用法如下：

当 $cwnd < ssthresh$ 时, 使用慢开始算法。

当 $cwnd > ssthresh$ 时, 改用拥塞避免算法。

当 $cwnd = ssthresh$ 时, 慢开始与拥塞避免算法任意。

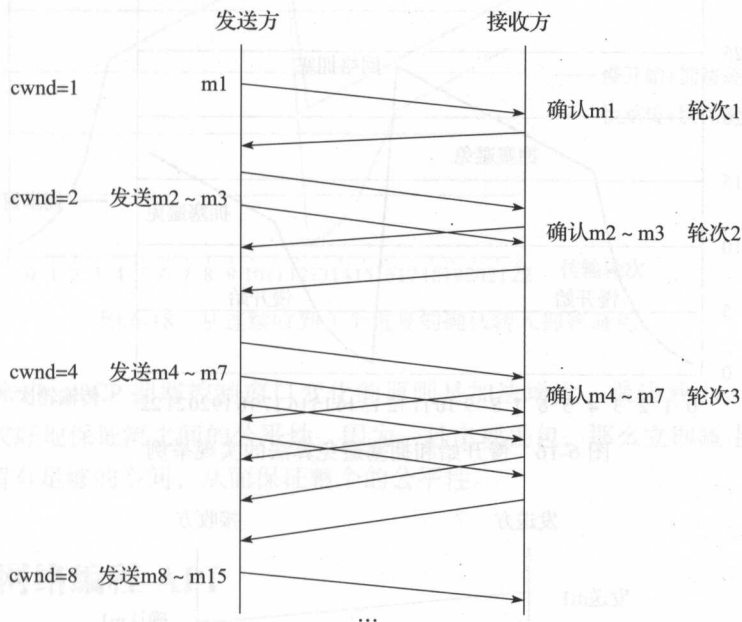


图 6-15 发送方每收到一个确认就把窗口 $cwnd+1$

拥塞避免算法让拥塞窗口缓慢增长, 即每经过一个往返时间 RTT 就把发送方的拥塞窗口 $cwnd$ 加 1, 而不是加倍。

拥塞控制具体过程如下所述。

- (1) TCP 连接初始化, 将拥塞窗口设置为 1。
- (2) 执行慢开始算法, $cwnd$ 按指数规律增长, 直到 $cwnd = ssthresh$ 时, 开始执行拥塞避免算法, $cwnd$ 按线性规律增长。

- (3) 当网络发生拥塞, 把 $ssthresh$ 值更新为拥塞前 $ssthresh$ 值的一半, $cwnd$ 重新设置为 1, 按照步骤 (2) 执行。

拥塞控制具体过程如图 6-16 所示。

再次提醒这里只是为了讨论方便而将拥塞窗口大小的单位改为数据报的个数, 实际上应当是字节数。

2. 快重传和快恢复

快重传要求接收方在收到一个失序的报文段后就立即发出重复确认 (为的是使发送方及早知道有报文段没有到达对方), 而不要等到自己发送数据时捎带确认。快重传算法规定, 发送方只要一连收到 3 个重复确认就应当立即重传对方尚未收到的报文段, 而不必继续等待设

置的重传计时器时间到期，如图 6-17 所示。



图 6-16 慢开始和拥塞避免算法的实现举例

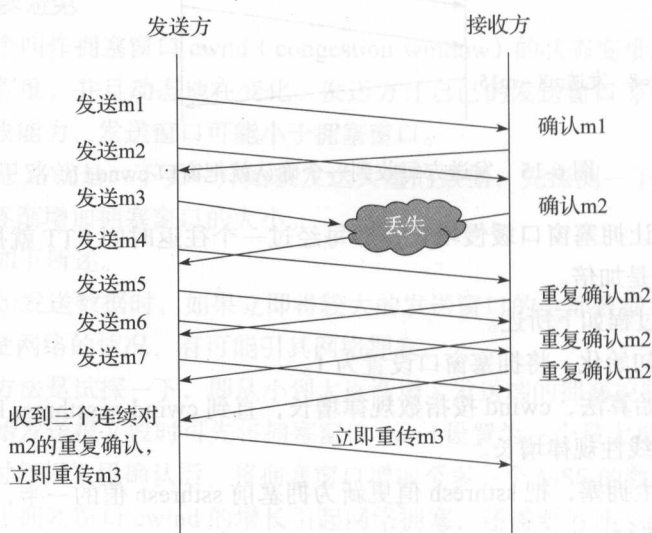


图 6-17 快重传的示意图

快重传配合使用的还有快恢复算法，有以下两个要点。

(1) 当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把 ssthresh 门限减半。但是接下去并不执行慢开始算法。

(2) 考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将 cwnd 设置为 ssthresh 的大小，然后执行拥塞避免算法，如图 6-18 所示。

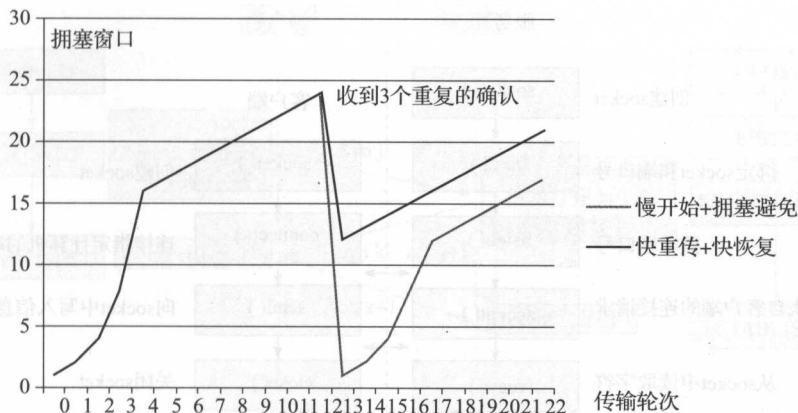


图 6-18 从连续收到 3 个重复的确认转入拥塞避免

从整体上来讲，TCP 拥塞控制窗口变化的原则是加法增大、乘法减小。可以看出 TCP 的该原则可以较好地保证流之间的公平性，因为一旦出现丢包，那么立即减半退避，可以给其他新建的流留有足够的空间，从而保证整个的公平性。

6.2 TCP 网络编程 API

网络中进程之间如何通信？首要解决的问题是如何唯一标识一个进程，否则通信无从谈起。在本地可以通过进程 PID 来唯一标识一个进程，但是在网络中这是行不通的。其实 TCP/IP 协议族已经帮我们解决了这个问题，网络层的 IP 地址可以唯一标识网络中的主机，而传输层的“协议 + 端口”可以唯一标识主机中的应用程序（进程）。这样利用三元组（IP 地址，协议，端口）就可以标识网络的进程了，网络中的进程通信就可以利用这个标志与其他进程进行交互。

上面我们已经知道网络中的进程是通过 socket 来通信的，那什么是 socket 呢？socket 起源于 UNIX，而 UNIX/Linux 基本哲学之一就是“一切皆文件”，都可以用“打开（open）→读写（write/read）→关闭（close）”模式来操作。socket 其实就是该模式的一个实现，socket 即是一种特殊的文件，一些 socket 函数就是对其进行的操作（读/写、打开、关闭），这些函数将在后面进行介绍。

使用 TCP/IP 协议的应用程序通常采用应用编程接口：UNIX BSD 的套接字（socket）和 UNIX System V 的 TLI（已经被淘汰），来实现网络进程之间的通信。就目前而言，几乎所有的应用程序都是采用 socket，而现在又是网络时代，网络中进程通信是无处不在，这就是“一切皆 socket”。

既然 socket 是“open—write/read—close”模式的一种实现，那么 socket 就提供了这些操作对应的函数接口。以 TCP 协议通信的 socket 为例，其交互流程大概如图 6-19 所示。

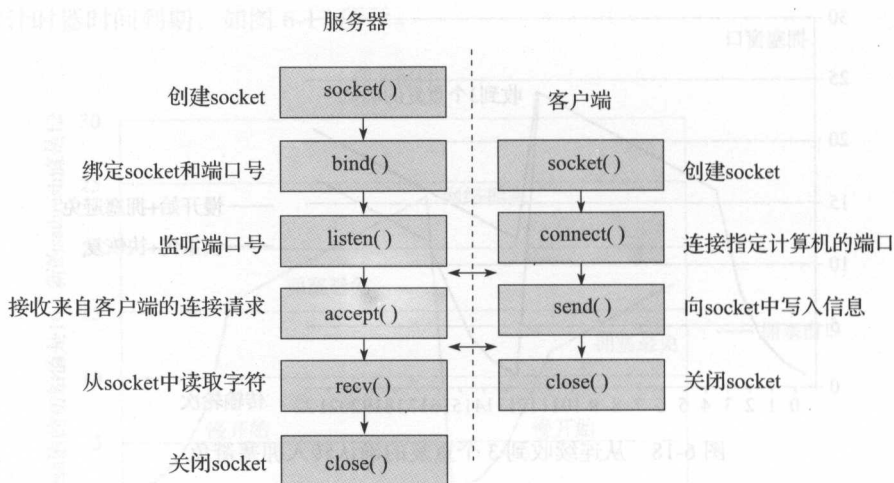


图 6-19 TCP 交互流程

图 6-19 中展示的交互流程，具体如下所述。

- (1) 服务器根据地址类型 (ipv4, ipv6)、socket 类型、协议创建 socket。
- (2) 服务器为 socket 绑定 IP 地址和端口号。
- (3) 服务器 socket 监听端口号请求，随时准备接收客户端发来的连接，这时候服务器的 socket 并没有被打开。
- (4) 客户端创建 socket。
- (5) 客户端打开 socket，根据服务器 IP 地址和端口号试图连接服务器 socket。
- (6) 服务器 socket 接收到客户端 socket 请求，被动打开，开始接收客户端请求，直到客户端返回连接信息。这时候 socket 进入阻塞状态，所谓阻塞即 accept() 方法一直到客户端返回连接信息后才返回，开始接收下一个客户端连接请求。
- (7) 客户端连接成功，向服务器发送连接状态信息。
- (8) 服务器 accept 方法返回，连接成功。
- (9) 客户端向 socket 写入信息。
- (10) 服务器读取信息。
- (11) 客户端关闭。
- (12) 服务器端关闭。

仔细一看，服务器 socket 和客户端 socket 建立连接的部分其实就是大名鼎鼎的 3 次握手，如图 6-20 所示，详细分析可参考 6.1.3 节内容。

下面介绍几个基本的 socket 接口函数。

1. socket 函数

socket 的函数原型如下所示：

```
int socket(int domain, int type, int protocol);
```

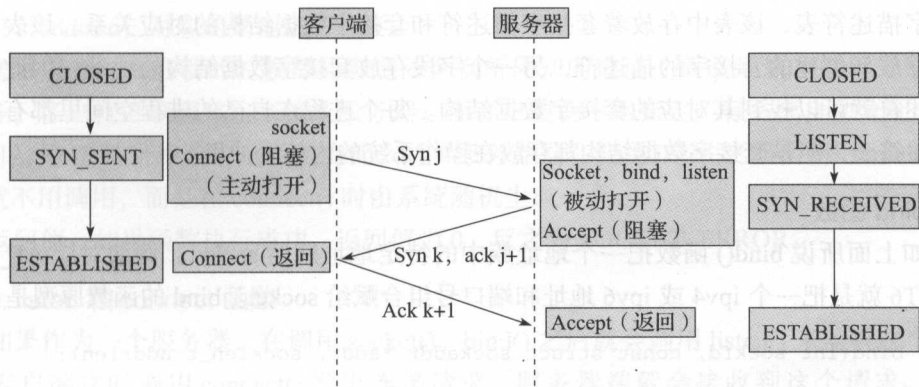


图 6-20 TCP 三次握手

socket 函数对应于普通文件的打开操作。普通文件的打开操作返回一个文件描述字，而 socket() 用于创建一个 socket 描述符 (socket descriptor)，它唯一标识一个 socket。这个 socket 描述字跟文件描述字一样，后续的操作都有用到它，把它作为参数，通过它来进行一些读写操作。

正如可以给 fopen 的传入不同参数值，以打开不同的文件一样，创建 socket 时，也可以指定不同的参数创建不同的 socket 描述符，socket 函数的 3 个参数分别如下所述。

(1) domain：即协议域，又称为协议族 (family)。常用的协议族有：AF_INET、AF_INET6、AF_LOCAL (或称 AF_UNIX，Unix 域 socket)、AF_ROUTE 等。协议族决定了 socket 的地址类型，在通信中必须采用对应的地址，如 AF_INET 决定了要用 ipv4 地址 (32 位) 与端口号 (16 位) 的组合、AF_UNIX 决定了要用一个绝对路径名作为地址。

(2) type：指定 socket 类型。常用的 socket 类型有：SOCK_STREAM、SOCK_DGRAM、SOCK_RAW、SOCK_PACKET、SOCK_SEQPACKET 等。其中，SOCK_STREAM 表示提供面向连接的稳定数据传输，即 TCP 协议。SOCK_DGRAM 表示使用不连续、不可靠的数据包连接。

(3) protocol：指定协议。常用的协议有，IPPROTO_TCP、IPPROTO_UDP、IPPROTO_SCTP、IPPROTO_TIPC 等，它们分别对应 TCP 传输协议、UDP 传输协议、STCP 传输协议、TIPC 传输协议。



注意 并不是说上面的 type 和 protocol 是可以随意组合的，如 SOCK_STREAM 就不可以跟 IPPROTO_UDP 组合。当 protocol 为 0 时，会自动选择 type 类型对应的默认协议。

当调用 socket 创建一个 socket 时，返回的 socket 描述字它存在于协议族 (address family, AF_XXX) 空间中，但没有一个具体的地址。如果想要给它赋予一个地址，就必须调用 bind() 函数，否则系统就在调用 connect()、listen() 时自动随机分配一个端口。

如果调用成功就返回新创建的套接字的描述符，如果失败就返回 INVALID_SOCKET (Linux 下失败返回 -1)。套接字描述符是一个整数类型的值。每个进程的进程空间里都有一

个套接字描述符表，该表中存放着套接字描述符和套接字数据结构的对应关系。该表中有一个字段存放新创建的套接字的描述符，另一个字段存放套接字数据结构的地址，因此根据套接字描述符就可以找到其对应的套接字数据结构。每个进程在自己的进程空间里都有一个套接字描述符表，但是套接字数据结构都存放在操作系统的内核缓冲里。

2. bind 函数

正如上面所说 bind() 函数把一个地址族中的特定地址赋给 socket。例如对应 AF_INET、AF_INET6 就是把一个 ipv4 或 ipv6 地址和端口号组合赋给 socket。bind 的函数原型是：

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

函数的 3 个参数分别如下所述。

(1) sockfd：即 socket 描述字，它是通过 socket() 函数创建来唯一标识一个 socket 的。bind() 函数就是将给这个描述字绑定一个名字。

(2) addr：一个 const struct sockaddr* 指针，指向要绑定给 sockfd 的协议地址。这个地址结构根据地址创建 socket 时的地址协议族的不同而不同，如 ipv4 对应的是如下所示的代码：

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr  sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    uint32_t        s_addr;     /* address in network byte order */
};
```

ipv6 对应的代码如下：

```
struct sockaddr_in6 {
    sa_family_t    sin6_family; /* AF_INET6 */
    in_port_t      sin6_port;   /* port number */
    uint32_t       sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;   /* IPv6 address */
    uint32_t       sin6_scope_id; /* Scope ID (new in 2.4) */
};

struct in6_addr {
    unsigned char  s6_addr[16]; /* IPv6 address */
};
```

UNIX 域对应的代码如下：

```
#define UNIX_PATH_MAX 108
struct sockaddr_un {
    sa_family_t sun_family; /* AF_UNIX */
    char sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

(3) `addrlen`: 对应的是地址的长度。

通常服务器在启动的时候都会绑定一个众所周知的地址(如 ip 地址 + 端口号), 用于提供服务, 客户就可以通过它来接连服务器; 而客户端就不用指定, 有系统自动分配一个端口号和自身的 IP 地址组合。这就是为什么通常服务器端在调用 `listen` 之前会调用 `bind()`; 而客户端就不用调用, 而是在 `connect()` 时由系统随机生成一个。

返回值: 如果函数执行成功, 返回值为 0, 反之为 `SOCKET_ERROR`。

3. `listen` 和 `connect` 函数

如果作为一个服务器, 在调用 `socket()`、`bind()` 之后就会调用 `listen()` 来监听这个 `socket`, 如果客户端这时调用 `connect()` 发出连接请求, 服务器端就会接收到这个请求。`listen` 和 `connect` 的函数原型是:

```
int listen(int sockfd, int backlog);
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

`listen` 函数的第一个参数即为要监听的 `socket` 描述字, 第二个参数为相应 `socket` 可以排队的最连接个数。`socket()` 函数创建的 `socket` 默认是一个主动类型的, `listen` 函数将 `socket` 变为被动类型的, 等待客户的连接请求。

`connect` 函数的第一个参数即为客户端的 `socket` 描述字, 第二参数为服务器的 `socket` 地址, 第三个参数为 `socket` 地址的长度。客户端通过调用 `connect` 函数来建立与 TCP 服务器的连接。

4. `accept` 函数

TCP 服务器端依次调用 `socket()`、`bind()`、`listen()` 之后, 就会监听指定的 `socket` 地址了。TCP 客户端依次调用 `socket()`、`connect()` 之后就会向 TCP 服务器发送了一个连接请求。TCP 服务器监听到这个请求之后, 就会调用 `accept()` 函数取接收请求, 这样连接就建立好了。之后就可以开始网络 I/O 操作了, 即类同于普通文件的读写 I/O 操作。`accept` 的函数原型是:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

`accept` 函数的第一个参数为服务器的 `socket` 描述字; 第二个参数为指向 `struct sockaddr*` 的指针, 用于返回客户端的协议地址; 第三个参数为协议地址的长度。如果 `accept` 成功, 那么其返回值是由内核自动生成的一个全新的描述字, 代表与返回客户的 TCP 连接。

注意: `accept` 的第一个参数为服务器的 `socket` 描述字, 是服务器开始调用 `socket()` 函数生成的, 称为监听 `socket` 描述字; 而 `accept` 函数返回的是已连接的 `socket` 描述字。一个服务器通常仅仅只创建一个监听 `socket` 描述字, 它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户创建了一个已连接 `socket` 描述字, 当服务器完成了对某个客户的服务, 相应的已连接 `socket` 描述字就被关闭。

5. read 和 write 函数

至此服务器与客户已经建立好连接了，可以调用网络 I/O 进行读写操作了，即实现了网络中不同进程之间的通信。网络 I/O 操作有下面几组：

```
read()/write()
recv()/send()
readv()/writev()
recvmsg()/sendmsg()
recvfrom()/sendto()
```

最常用的则是 read() 和 write()。read() 的函数原型是：

```
ssize_t read(int fd, void *buf, size_t count);
```

read() 函数是负责从 fd 中读取内容。当读取成功时，read() 返回实际所读的字节数，如果返回的值是 0 表示已经读到文件的结束了，小于 0 表示出现了错误。如果错误为 EINTR 说明读是由中断引起的，如果是 ECONNRESET 表示网络连接出了问题。3 个参数分别是：

① socket 描述字 fd；② 缓冲区 buf；③ 缓冲区长度 count。

write() 的函数原型是：

```
ssize_t write(int fd, const void *buf, size_t count);
```

write() 函数将 buf 中的 nbytes 字节内容写入文件描述符 fd 成功时返回写的字节数。失败时返回 -1，并设置 errno 变量。在网络程序中，当我们向套接字文件描述符写时有两种可能：① write 的返回值大于 0，表示写了部分或者是全部的数据；② 返回的值小于 0，此时出现了错误。实际中要根据错误类型来处理。如果错误为 EINTR 表示在写的时候出现了中断错误。如果为 EPIPE 表示网络连接出现了问题（对方已经关闭了连接）。3 个参数分别是：① fd 表示 socket 描述字；② buf 表示缓冲区；③ count 表示缓冲区长度。

6. close 函数

在服务器与客户端建立连接之后，会进行一些读写操作，完成了读写操作就要关闭相应的 socket 描述字，好比操作完打开的文件要调用 close 关闭打开的文件。需要包含的头文件是：

```
#include <unistd.h>
```

close 的函数原型是：

```
int close(int fd);
```

close 一个 TCP socket 的默认行为时，会把该 socket 标记为以关闭，然后立即返回到调用进程。该描述字不能再由调用进程使用，也就是说不能再作为 read 或 write 的第一个参数。



注意 close 操作只是使相应 socket 描述字的引用计数 -1，只有当引用计数为 0 的时候，才会触发 TCP 客户端向服务器发送终止连接请求。

6.3 实现一个 TCP server

下面用 TCP 协议编写一个简单的服务器、客户端，其中服务器端一直监听本机的 6666 号端口。如果收到连接请求，将接收请求并接收客户端发来的消息；客户端与服务器端建立连接并发送一条消息。

【例 6.1】一个简单的服务器和客户端。

server.cpp 的代码是：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>

#define MAXLINE 4096

int main(int argc, char** argv){
    int listenfd, connfd;
    struct sockaddr_in servaddr;
    char buff[4096];
    int n;

    if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ){
        printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
        return 0;
    }

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(6666);

    if( bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1){
        printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);
        return 0;
    }

    if( listen(listenfd, 10) == -1){
        printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);
        return 0;
    }

    printf("====waiting for client's request====\n");
    while(1){
        if( (connfd = accept(listenfd, (struct sockaddr*)NULL, NULL)) == -1){
```

```

    printf("accept socket error: %s(errno: %d)",strerror(errno),errno);
    continue;
}
n = recv(connfd, buff, MAXLINE, 0);
buff[n] = '\0';
printf("recv msg from client: %s\n", buff);
close(connfd);
}
close(listenfd);
return 0;
}

```

client.cpp 的代码是:

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#define MAXLINE 4096

int main(int argc, char** argv){
    int sockfd, n;
    char recvline[4096], sendline[4096];
    struct sockaddr_in servaddr;

    if( argc != 2){
        printf("usage: ./client <ipaddress>\n");
        return 0;
    }

    if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        printf("create socket error: %s(errno: %d)\n", strerror(errno),errno);
        return 0;
    }

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(6666);
    if( inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0){
        printf("inet_pton error for %s\n",argv[1]);
        return 0;
    }

    if( connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0){
        printf("connect error: %s(errno: %d)\n",strerror(errno),errno);
        return 0;
    }
}

```



```

printf("send msg to server: \n");
fgets(sendline, 4096, stdin);
if( send(sockfd, sendline, strlen(sendline), 0) < 0){
    printf("send msg error: %s(errno: %d)\n", strerror(errno), errno);
    return 0;
}
close(sockfd);
return 0;
}

```

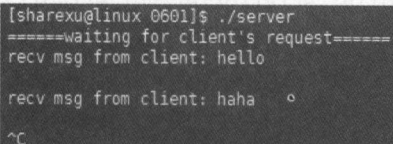
makefile 的代码是:

```

all:server client
server:server.o
    g++ -g -o server server.o
client:client.o
    g++ -g -o client client.o
server.o:server.cpp
    g++ -g -c server.cpp
client.o:client.cpp
    g++ -g -c client.cpp
clean:all
    rm all

```

执行 make 命令后, 生成 server 和 client2 个可执行文件。分别打开两个终端窗口, 一个执行 ./server 命令, 一个执行 ./client 127.0.0.1 命令, 表示连上本机的 6666 端口, 执行 ./server 命令的要先执行。执行 ./client 127.0.0.1 命令后, 会提示说要发给 server 的内容, 输入 “hello” 后, client 客户端执行完毕, 这时可以看到 server 的那个终端窗口输出 “recv msg from client: hello”。继续执行 ./client 127.0.0.1 命令后, 再输入 “haha”, server 的终端继续输出 “recv msg from client: haha”。结果如图 6-21 和图 6-22 所示。

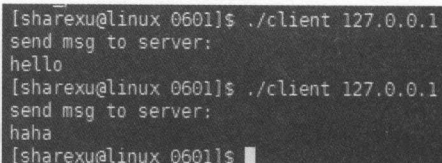


```

[sharexu@linux 0601]$ ./server
=====waiting for client's request=====
recv msg from client: hello
recv msg from client: haha
^C

```

图 6-21 例 6.1 服务端执行结果图



```

[sharexu@linux 0601]$ ./client 127.0.0.1
send msg to server:
hello
[sharexu@linux 0601]$ ./client 127.0.0.1
send msg to server:
haha
[sharexu@linux 0601]$

```

图 6-22 例 6.1 客户端执行结果图

接着来解释下 server.cpp 和 client.cpp 中分别做了些什么。

server.cpp 中创建了一个 TCP 的 socket, 并打印了如果创建失败时的错误码如下所示:

```

if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ){
    printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
    exit(0);
}

```

可以在一个终端执行 `./server` 后，再在另一个终端里执行 `./server`，发现第二个终端里输出了如下所示的提示：

```
[sharexu@linux 0601]$ ./server
bind socket error: Address already in use(errno: 98)
[sharexu@linux 0601]$
```

该地址已经被使用，无法再创建 socket。不仅打印了错误码，还打印出了错误信息。`strerror (errno)` 就可以打印错误信息。先把 `servaddr` 地址清空，再赋值，代码是：

```
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(6666);
```

`INADDR_ANY` 就是指定地址为 `0.0.0.0` 的地址，这个地址事实上表示不确定地址，或“所有地址”“任意地址”。一般来说，在各个系统中均定义成为 0 值。

`INADDR_ANY` 是用于多网卡的机器上的，多网卡就会有多个 IP 机器。比如你的机器有 3 个 IP：192.168.1.1、202.202.202.202 和 61.1.2.3。如果设置 `serv.sin_addr.s_addr=inet_addr("192.168.1.1")`；然后监听 100 端口，这时其他机器只有连接到 192.168.1.1:100 才能成功；连接 202.202.202.202:100 或 61.1.2.3:100 都会失败。如果设置 `serv.sin_addr.s_addr=htonl(INADDR_ANY)`；的话，无论连接哪个 IP 都可以连上的。

接下来就是将地址绑定到 `listenfd` 上，代码为：

```
if( bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1){
    printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);
    exit(0);
}
```

监听这个 `listenfd`，代码为：

```
if( listen(listenfd, 10) == -1){
    printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);
    exit(0);
}
```

在 `while` 循环里持续接收包，注意，`accept` 和 `recv` 是都是在 `while` 循环里的，也就是收到包之后，这个 `fd` 就没用了，并关闭它，下一个包重新接收包。注意，`recv` 函数接收到的字符串是不带 `"\0"` 结束符的，要用 `printf` 输出时必须得先加上结束符 `"\0"`，如下所示。

```
while(1){
    if( (connfd = accept(listenfd, (struct sockaddr*)NULL, NULL)) == -1){
        printf("accept socket error: %s(errno: %d)",strerror(errno),errno);
        continue;
    }
```

```

n = recv(connfd, buff, MAXLINE, 0);
buff[n] = '\0';
printf("recv msg from client: %s\n", buff);
close(connfd);
}

```

客户端 `client.cpp` 的代码则简单得多，只需绑定 `socket` 后，直接 `connect` 后即可发包。注意，程序中将输入的 `ip` 地址用 `inet_pton` 函数进行了转换。`inet_pton` 是 `IP` 地址转换函数，可以在将 `IP` 地址在“点分十进制”和“二进制整数”之间转换。`inet_pton` 函数所需的头文件是：

```

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

```

`inet_pton` 的函数原型是：

```
int inet_pton(int af, const char *src, void *dst);
```

这个函数转换字符串到网络地址，第一个参数 `af` 是地址族，转换后存在 `dst` 中。

`inet_pton` 是 `inet_addr` 的扩展，支持的多地址族有下面几种。

1) `af = AF_INET`，`src` 为指向字符型的地址，即 `ASCII` 的地址的首地址（`ddd.ddd.ddd.ddd` 格式的），函数将该地址转换为 `in_addr` 的结构体，并复制在 `*dst` 中。

2) `af = AF_INET6`，`src` 为指向 `IPV6` 的地址，函数将该地址转换为 `in6_addr` 的结构体，并复制在 `*dst` 中。

如果函数出错将返回一个负值，并将 `errno` 设置为 `EAFNOSUPPORT`，如果参数 `af` 指定的地址族和 `src` 格式不对，函数将返回 0。

6.4 TCP 协议选项

`TCP` 头部的选项部分是为了 `TCP` 适应复杂的网络环境和更好地服务于应用层而进行设计的。`TCP` 选项部分最长可以达到 40 Byte，再加上 `TCP` 选项外的固定的 20 Byte 部分，`TCP` 的最长头部可达 60 Byte。`TCP` 头部长度可以通过 `TCP` 头部中的“数据偏移”位来查看。值得注意的是：`TCP` 偏移量的单位是 32bit，也就是 4 Byte。而 `TCP` 偏移量共占 4bit，取最大的 1111 (B) 计算也就是十进制的 15。15*4 Byte=60 Byte，这个也是 `TCP` 的首部不超过 60 Byte 的原因。

`TCP` 选项部分实际运用的有以下几种。

(1) `SO_REUSEADDR`。

(2) `SO_RECVBUF/SO_SNDBUF`。

(3) `SO_KEEPALIVE`。

- (4) SO_LINGER。
- (5) TCP_CORK。
- (6) TCP_NODELAY。
- (7) TCP_DEFER_ACCEPT。
- (8) TCP_KEEPCNT、TCP_KEEPIDLE 和 TCP_KEEPINTVL。
- (9) SO_SNDTIMEO 和 SO_RCVTIMEO。
- (10) SO_RCVBUF 和 SO_SNDBUF。

1. SO_REUSEADDR

一般来说，一个端口释放后会等待两分钟左右后才能再被使用，而使用 SO_REUSEADDR 则可以让端口释放后立即就可以被再次使用。SO_REUSEADDR 用于对 TCP 套接字处于 TIME_WAIT 状态下的 socket，才可以重复绑定使用。server 程序总是应该在调用 bind() 之前设置 SO_REUSEADDR 套接字选项。TCP 先调用 close() 的一方会进入 TIME_WAIT 状态。

SO_REUSEADDR 提供了以下 4 个功能。

1) SO_REUSEADDR 允许启动一个监听服务器并捆绑其众所周知的端口，并且以前建立的将此端口用做它们的本地端口的连接仍存在。这通常是重启监听服务器时会出现的情况，若不设置此选项，则 bind 时将出错。

2) SO_REUSEADDR 允许在同一端口上启动同一服务器的多个实例，只要每个实例捆绑一个不同的本地 IP 地址即可。对于 TCP，根本不可能同时启动捆绑相同 IP 地址和相同端口号的多个服务器。

3) SO_REUSEADDR 允许单个进程捆绑同一端口到多个套接口上，只要每个进程捆绑的指定不同的本地 IP 地址即可。

4) SO_REUSEADDR 允许完全重复的捆绑：当一个 IP 地址和端口绑定到某个套接口上时，还允许此 IP 地址和端口捆绑到另一个套接口上。一般来说，这个特性仅支持多播的系统上才有，而且只对 UDP 套接口而言（TCP 不支持多播）。

SO_REUSEPORT 和 SO_REUSEADDR 有类似的功能，SO_REUSEPORT 选项有如下语义。

1) 此选项允许完全重复捆绑，但只有在想捆绑相同 IP 地址和端口的套接口都指定了此套接口选项才行。

2) 如果被捆绑的 IP 地址是一个多播地址，则 SO_REUSEADDR 和 SO_REUSEPORT 等效。

在所有 TCP 服务器中，在调用 bind 之前设置 SO_REUSEADDR 套接口选项；当编写一个同一时刻在同一主机上可运行多次的多播应用程序时，设置 SO_REUSEADDR 选项，并将本组的多播地址作为本地 IP 地址捆绑。用法如下所示：

```
if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, (const void *)&nOptval,
sizeof(int)) < 0)
```

在编写 TCP/SOCK_STREAM 服务程序时, SO_REUSEADDR 这个套接字选项会通知内核, 如果端口忙, 但 TCP 状态位于 TIME_WAIT 状态, 则可以重用端口; 如果端口忙, 而 TCP 状态位于其他状态, 重用端口时依旧会得到一个错误信息, 指明“地址已经使用中”。如果服务程序停止后想立即重启, 而新套接字依旧使用同一端口, 此时使用 SO_REUSEADDR 选项非常有用。但必须意识到, 此时任何非期望数据到达, 都可能导致服务程序反应混乱, 不过这只是一种可能, 事实上可能性较小。一个套接字由相关五元组构成: 协议、本地地址、本地端口、远程地址、远程端口。SO_REUSEADDR 仅仅表示可以重用本地地址、本地端口, 整个相关五元组还是唯一确定的。所以, 重启后的服务程序有可能收到非期望数据。综上所述, 必须慎重使用 SO_REUSEADDR 选项。

2. TCP_NODELAY/TCP_CHORK

在网络拥塞控制领域, 有一个非常有名的算法叫作 Nagle 算法 (Nagle algorithm), 这是使用它的发明人 John Nagle 的名字来命名的, John Nagle 在 1984 年首次用这个算法来尝试解决福特汽车公司的网络拥塞问题 (RFC 896)。该问题的具体描述是: 如果应用程序一次产生 1 Byte 的数据, 而这个 1 Byte 数据又以网络数据包的形式发送到远端服务器, 那么就很容易导致网络由于太多的数据包而超载。比如, 当用户使用 Telnet 连接到远程服务器时, 每一次按键操作就会产生 1 Byte 数据, 进而发送出去一个数据包, 所以在典型情况下, 传送一个只拥有 1 Byte 有效数据的数据包, 却要发费 40 Byte 长包头 (即 IP 头 20 Byte+TCP 头 20 Byte) 的额外开销, 这种有效载荷 (payload) 利用率极其低下的情况被统称之为愚蠢窗口症候群 (Silly Window Syndrome)。可以看到, 这种情况对于轻负载的网络来说, 可能还可以接受, 但是对于重负载的网络而言, 就极有可能承载不了而轻易发生拥塞瘫痪。

针对上面提到的这个状况, Nagle 算法的改进在于: 如果发送端欲多次发送包含少量字符的数据包 (一般情况下, 后面统一称长度小于 MSS 的数据包为小包; 与此相对, 称长度等于 MSS 的数据包为大包; 为了某些对比说明, 还有中包, 即长度比小包长, 但又不足一个 MSS 的包), 则发送端会先将第一个小包发送出去, 而将后面到达的少量字符数据都缓存起来而不立即发送, 直到收到接收端对前一个数据包报文段的 ACK 确认为止, 或当前字符属于紧急数据, 或者积攒到了一定数量的数据 (比如缓存的字符数据已经达到数据包报文段的最大长度) 等多种情况才将其组成一个较大的数据包发送出去。

TCP 中的 Nagle 算法默认是启用的, 但是它并不是适合任何情况, 对于 telnet 或 rlogin 这样的远程登录应用的确比较适合 (原本就是为此而设计), 但是在某些应用场景下却又需要关闭它, 有需要发送的就立即发送。

TCP_NODELAY 和 TCP_CORK 基本上控制了包的“Nagle 化”, Nagle 化在这里的含义是采用 Nagle 算法把较小的包组装为更大的帧。TCP_NODELAY 和 TCP_CORK 都禁掉了

Nagle 算法，只不过它们的行为需求不同而已。

TCP_NODELAY 不使用 Nagle 算法，不会将小包进行拼接成大包再进行发送，而是直接将小包发送出去，这会使得用户体验要好。

当在传送大量数据的时候，为了提高 TCP 发送效率，可以设置 TCP_CORK，CORK 就是“塞子”的意思，它会尽量在每次发送最大的数据量。当设置了 TCP_CORK 后，会有 200ms 阻塞，当阻塞时间过后，数据就会自动传送。

3. SO_LINGER

linger 是延迟延缓的意思，这里的延缓是指面向连接的 socket 的 close 操作。默认 close 立即返回，但是当发送缓冲区中还有一部分数据的时候，系统将会尝试将数据发送给对端。

SO_LINGER 可以改变 close 的行为。控制 SO_LINGER 通过下面一个结构：

```
struct linger{
    int l_onoff; /*0=off, nonzero=on*/
    int l_linger; /*linger time, POSIX specifies units as seconds*/
};
```

(1) 通过结构体中成员的不同赋值，可以表现为下面几种情况。

1) l_onoff 设置为 0，选项被关闭。l_linger 值被忽略，就是上面的默认情形，close 立即返回。

2) l_onoff 设置为非 0，l_linger 被设置为 0，则 close() 不被阻塞立即执行，丢弃 socket 发送缓冲区中的数据，并向对端发送一个 RST 报文。这种关闭方式称为“强制”或“失效”关闭。这种方式，是用非正常的 4 种握手方式结束 TCP 的连接，所以，TCP 连接将不会进入 TIME_WAIT 状态，这样会导致新建的可能和已经连接的数据造成混乱。

3) l_onoff 设置为非 0，l_linger 被设置为非 0，则 close() 调用阻塞进程，直到所剩数据发送完毕或超时。这种关闭称为“优雅地”关闭，在这种情况下，close 的返回得到延迟。调用 close 去关闭 socket 的时候，内核将会延迟。也就是说，如果 send buffer 中还有数据尚未发送，该进程将会被休眠直到以下任何一种情况发生：① send buffer 中的所有数据都被发送并且得到对方 TCP 的应答消息（这种应答并不是意味着对方应用程序已经接收到数据，在后面 shutdown 中将会具体讲解）；②延迟时间消耗完，在延迟时间被消耗完之后，send buffer 中的所有数据都将会被丢弃。

上面①、②两种情况中，如果 socket 被设置为 O_NONBLOCK 状态，程序将不会等待 close 返回，send buffer 中的所有数据都将会被丢弃。所以需要判断 close 的返回值。在 send buffer 中的所有数据都被发送之前并且延迟时间没有消耗完，close 返回的话，close 将会返回一个 EWOULDBLOCK 的提示。



注意 这个选项需要谨慎使用，尤其是强制式关闭，可能会丢失服务器发给客户端的最后一部分数据。

(2) 实例说明。

1) close 默认操作：立即返回。

此种情况，close 立即返回，如果 send buffer 中还有数据，close 将会等到所有数据被发送完之后返回。由于并没有等待对方 TCP 发送的 ACK 信息，所以只能保证数据已经发送到对方，所以并不知道对方是否已经接受了数据。由于此种情况，TCP 连接终止是按照正常的 4 次挥手方式，需要经过 TIME_WAIT，默认情况下数据流向如图 6-23 所示。

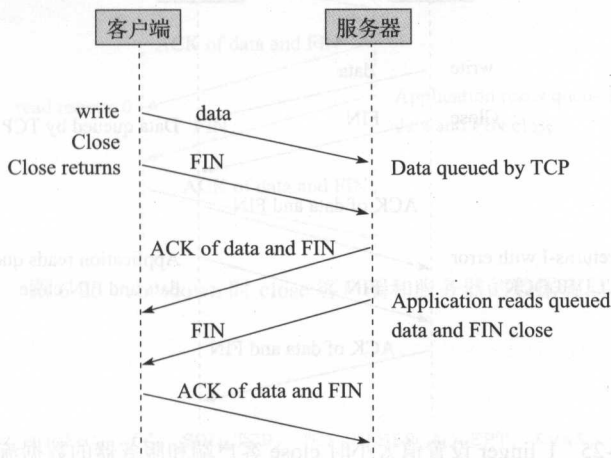


图 6-23 默认 close 时客户端和服务器的数据流向

2) l_onoff 非 0，并且使之 l_linger 为一个整数。

在这种情况下，close 会在接收到对方 TCP 的 ACK 信息之后才返回（l_linger 消耗完之前）。但是这种 ACK 信息只能保证对方已经接收到数据，并不保证对方应用程序已经读取数据，如图 6-24 所示。

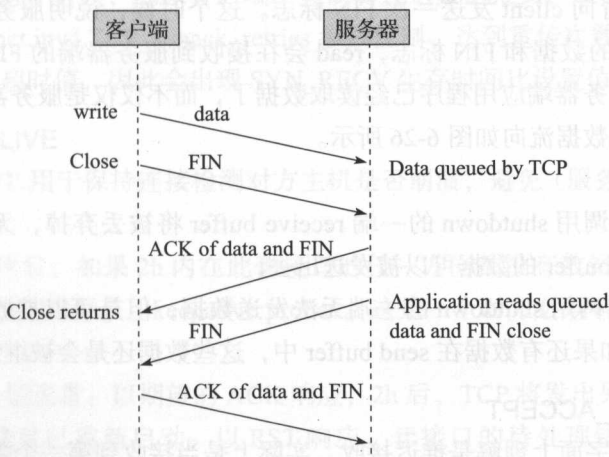


图 6-24 l_onoff 和 l_linger 都非 0 时 close 客户端和服务器的数据流向

3) `l_linger` 设置值太小。

这种情况，由于 `l_linger` 值太小，在 `send buffer` 中的数据都发送完之前，`close` 就返回，此种情况终止 TCP 连接，更 `l_linger = 0` 类似，TCP 连接终止不是按照正常的 4 步握手，所以，TCP 连接不会进入 `TIME_WAIT` 状态，那么，client 会向 server 发送一个 `RST` 信息，如图 6-25 所示。

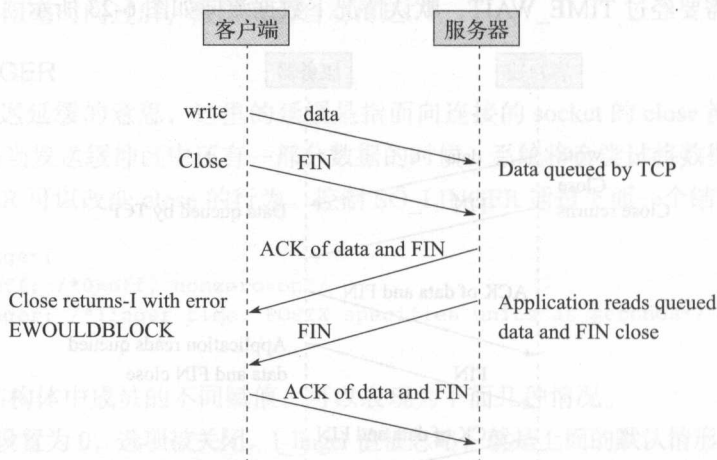


图 6-25 `l_linger` 设置值太小时 `close` 客户端和服务器的数据流向

4) `shutdown` 等待应用程序读取数据。

同上面的 2) 进行对比，调用 `shutdown` 后紧接着调用 `read`，此时 `read` 会被阻塞，直到接收到对方的 `FIN` 标志，也就是说 `read` 是在服务器端的应用程序调用 `close` 之后才返回的。当服务器端应用程序读取到来自 client 的数据和 `FIN` 标志之后，服务器端会进入 `CLOSE_WAIT` 状态，那么，如果此时服务器端要断开该 TCP 连接，需要服务器端应用程序调用一次 `close`，也就意味着向 client 发送一次 `FIN` 标志。这个时候，说明服务器端的应用程序已经读取到 client 发送的数据和 `FIN` 标志，`read` 会在接收到服务器端的 `FIN` 之后返回。所以，`shutdown` 可以确保服务器端应用程序已经读取数据了，而不仅仅是服务器端已经接收到数据而已，`shutdown` 时的数据流向如图 6-26 所示。

`shutdown` 参数有：

① `SHUT_RD` 指调用 `shutdown` 的一端 `receive buffer` 将被丢弃掉，无法接收数据，但是可以发送数据，`send buffer` 的数据可以被发送出去；

② `SHUT_WR` 指调用 `shutdown` 的一端无法发送数据，但是可以接收数据。该参数表示不能调用 `send`。但是如果还有数据在 `send buffer` 中，这些数据还是会被继续发送出去的。

4. `TCP_DEFER_ACCEPT`

`defer accept`，从字面上理解是推迟接收，实际上是当接收到第一个数据之后，才会创建连接。对于像 HTTP 等非交互式的服务器，这个很有意义，可以用来防御空连接攻击（只是

建立连接，但是不发送任何数据)。

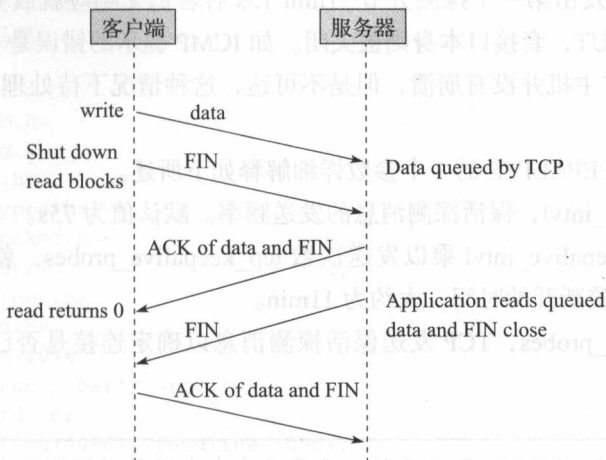


图 6-26 shutdown 时 close 客户端和服务器的数据流向

使用方法如下：

```
val = 5;
setsockopt(srv_socket->fd, SOL_TCP, TCP_DEFER_ACCEPT, &val, sizeof(val));
```

里面 val 的单位是秒 (s)，注意如果打开这个功能，kernel 在 val 时间之内还没有收到数据，不会继续唤醒进程，而是直接丢弃连接。如果服务器设置 TCP_DEFER_ACCEPT 选项后，服务器受到一个 CONNECT 请求并进行了 3 次握手之后，新的 socket 状态依然为 SYN_RECV，而不是 ESTABLISHED，操作系统不会接收数据。

由于设置 TCP_DEFER_ACCEPT 选项之后，3 次握手后状态没有达到 ESTABLISHED，而是 SYN_RECV。这个时候，如果客户端一直没有发送数据报文，服务器将重传 SYN/ACK 报文，重传次数受 net.ipv4.TCP_synack_retries 参数控制，达到重传次数之后，才会再次进行 setsockopt 中设置的超时值，因此会出现 SYN_RECV 生存时间比设置值大一些的情况。

5. SO_KEEPALIVE

SO_KEEPALIVE 用于保持连接检测对方主机是否崩溃，避免 (服务器) 永远阻塞于 TCP 连接的输入。

(1) 设置该选项后，如果 2h 内在此套接口的任一方向都没有数据交换，TCP 就自动给对方发一个保持存活探测分节 (keepalive probe)。这是一个对方必须响应的 TCP 分节，它会导致以下 3 种情况。

- 1) 对方接收一切正常：以期望的 ACK 响应，2h 后，TCP 将发出另一个探测分节。
- 2) 对方已崩溃且已重新启动：以 RST 响应。套接口的待处理错误被置为 ECONNRESET，套接口本身则被关闭。

3) 对方无任何响应：源自 `berkeley` 的 TCP 发送另外 8 个探测分节，相隔 75s 一个，试图得到一个响应。在发出第一个探测分节 11min 15s 后若仍无响应就放弃。套接口的待处理错误被置为 `ETIMEOUT`，套接口本身则被关闭。如 ICMP 提示的错误是 `host unreachable`（主机不可达），说明对方主机并没有崩溃，但是不可达，这种情况下待处理错误被置为 `EHOST-UNREACH` 状态。

(2) 有关 `SO_KEEPALIVE` 的 3 个参数详细解释如下所述。

1) `tcp_keepalive_intvl`，保活探测消息的发送频率。默认值为 75s。

发送频率 `tcp_keepalive_intvl` 乘以发送次数 `tcp_keepalive_probes`，就得到了从开始探测直到放弃探测确定连接断开的时间，大约为 11min。

2) `tcp_keepalive_probes`，TCP 发送保活探测消息以确定连接是否已断开的次数，其默认值为 9（次）。



注意 只有设置了 `SO_KEEPALIVE` 套接口选项后才会发送保活探测消息。

3) `tcp_keepalive_time`，在 TCP 保活打开的情况下，最后一次数据交换到 TCP 发送第一个保活探测消息的时间，即允许的持续空闲时间，其默认值为 7200s（2h）。

设置 `SO_KEEPALIVE` 选项来开启 `KEEPALIVE`，然后通过 `TCP_KEEPIIDLE`、`TCP_KEEPIINTVL` 和 `TCP_KEEPCNT` 设置 `keepalive` 的开始时间、间隔、次数等参数，使用方法如下：

```
//Setting For KeepAlive
int keepalive = 1;
setsockopt(incomingsock, SOL_SOCKET, SO_KEEPALIVE, (void*)&keepalive, (socklen_t)
sizeof(keepalive));
int keepalive_time = 30;
setsockopt(incomingsock, IPPROTO_TCP, TCP_KEEPIIDLE, (void*)&keepalive_
time), (socklen_t)sizeof(keepalive_time));
int keepalive_intvl = 3;
setsockopt(incomingsock, IPPROTO_TCP, TCP_KEEPIINTVL, (void*)&keepalive_
intvl), (socklen_t)sizeof(keepalive_intvl));
int keepalive_probes = 3;
setsockopt(incomingsock, IPPROTO_TCP, TCP_KEEPCNT, (void*)&keepalive_
probes), (socklen_t)sizeof(keepalive_probes));
```

当然，也可以通过设置 `/proc/sys/net/ipv4/tcp_keepalive_time`、`tcp_keepalive_intvl` 和 `tcp_keepalive_probes` 等内核参数来达到目的，但是这样的话，会影响所有的 `socket`，因此建议使用 `setsockopt` 设置。

6. `SO_SNDTIMEO` 和 `SO_RCVTIMEO`

`SO_SNDTIMEO` 和 `SO_RCVTIMEO` 两项分别设置 `socket` 的发送和接收超时时间，它们都接收一个 `timeval` 结构作为参数，当 `timeval` 结构为 0 时，表示选项无效。接收超时会影响 `read`、`readv`、`recv`、`recvfrom` 和 `recvmsg` 的状态；发送超时会影响 `write`、`writv`、`send`、

sendto 和 sendmsg 的状态。

【例 6.2】超时时间设置举例。

client.cpp 的代码是：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#define MAXLINE 4096
int main(int argc, char** argv){
    int connfd, n;
    char recvline[4096], sendline[4096];
    struct sockaddr_in servaddr;
    if( argc != 2){
        printf("usage: ./client <ipaddress>\n");
        return 0;
    }
    if( (connfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        printf("create socket error: %s(errno: %d)\n", strerror(errno), errno);
        return 0;
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(6666);
    if( inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0){
        printf("inet_pton error for %s\n", argv[1]);
        return 0;
    }
    if(connect(connfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0){
        printf("connect error: %s(errno: %d)\n", strerror(errno), errno);
        return 0;
    }
    struct timeval stTimeValStruct;
    stTimeValStruct.tv_sec = 2;
    stTimeValStruct.tv_usec = 0;
    if(setsockopt(connfd, SOL_SOCKET, SO_SNDTIMEO, &stTimeValStruct, sizeof(stTimeValStruct)){
        printf("setsockopt error: %s(errno: %d)\n", strerror(errno), errno);
        return 0;
    }
    if(setsockopt(connfd, SOL_SOCKET, SO_RCVTIMEO, &stTimeValStruct, sizeof(stTimeValStruct)){
        printf("setsockopt error: %s(errno: %d)\n", strerror(errno), errno);
        return 0;
    }
}
```

```

    ssize_t writeLen;
    char sendMsg[10] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '\0' };
    int count = 0;
    writeLen = write(connfd, sendMsg, sizeof(sendMsg));
    if (writeLen < 0) {
        printf("write error: %s(errno: %d)\n", strerror(errno), errno);
        close(connfd);
        return 0;
    }
    else{
        printf("write sucess, writelen :%d, sendMsg:%s\n",writeLen,sendMsg);
    }
    char readMsg[10]={0};
    int readlen= read(connfd,readMsg,sizeof(readMsg));
    if (readlen < 0) {
        printf("read error: %s(errno: %d)\n", strerror(errno), errno);
        close(connfd);
        return 0;
    }
    else{
        readMsg[9]= '\0';
        printf("read sucess, readlen :%d, readMsg:%s\n",readlen,readMsg);
    }
    close(connfd);
    return 0;
}

```

server.cpp 的代码是：

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
#define MAXLINE 4096
int main(int argc, char** argv){
    int listenfd, acceptfd;
    struct sockaddr_in servaddr;
    if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ){
        printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
        return -1;
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(6666);
    if( bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1){
        printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);
        return -1;
    }
}

```

```

}
if( listen(listenfd, 10) == -1){
    printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);
    return -1;
}
printf("====waiting for client's request====\n");
if( (acceptfd = accept(listenfd, (struct sockaddr*)NULL, NULL)) == -1){
    printf("accept socket error: %s(errno: %d)",strerror(errno),errno);
}
char recvMsg[100];
ssize_t readLen = read(acceptfd, recvMsg, sizeof(recvMsg));
if (readLen < 0) {
    printf("read error: %s(errno: %d)\n",strerror(errno),errno);
    return -1;
}
recvMsg[9]='\\0';
printf("readLen:%d, recvMsg:%s\n" ,readLen,recvMsg);
sleep(5);
recvMsg[1]='9';
ssize_t writeLen = write(acceptfd, recvMsg, sizeof(recvMsg));
printf("writeLen:%d, sendMsg:%s\n" ,writeLen,recvMsg);
if (writeLen < 0) {
    printf("writeLen error: %s(errno: %d)\n",strerror(errno),errno);
    return -1;
}
close(acceptfd);
return 0;
}

```

makefile 的代码是:

```

all:server client
server:server.o
    g++ -g -o server server.o
client:client.o
    g++ -g -o client client.o
server.o:server.cpp
    g++ -g -c server.cpp
client.o:client.cpp
    g++ -g -c client.cpp
clean:all
    rm all

```

执行 make 命令后, 生成 server 和 client 2 个可执行文件, 分别打开两个终端窗口, 一个执行 ./server 命令, 一个执行 ./client 127.0.0.1 命令, 表示连上本机的 6666 端口, 并且 ./server 命令要先执行。执行 ./client 127.0.0.1 命令后, 会输出 “write sucess, writelen :10, sendMsg:012345678”, 表示已经发送了一个包给 server。此时 server 这边也输出了 “readLen:10, recvMsg:012345678”。过了 2s 之后, 发现 client 这边提示 “read error: Resource temporarily unavailable(errno: 11)”, 再过了 3s 之后, server 这边提示 “writeLen:100, sendMsg:092345678”。

执行结果如图 6-27 和图 6-28 所示。

```
[sharexu@linux 0602]$ ./client 127.0.0.1
write success, writelen:10, sendMsg:012345678
read error: Resource temporarily unavailable(errno: 11)
[sharexu@linux 0602]$
```

图 6-27 例 6.2 客户端执行结果图

```
[sharexu@linux 0602]$ ./server
=====waiting for client's request=====
readLen:10, recvMsg:012345678
writelen:100, sendMsg:092345678
[sharexu@linux 0602]$
```

图 6-28 例 6.2 服务端执行结果图

再来看下程序里分别都有哪些功能。

client.cpp 中，下面这些都为连上服务端 6666 端口的相关代码：

```
if( (connfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
    printf("create socket error: %s(errno: %d)\n", strerror(errno),errno);
    return 0;
}
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(6666);
if( inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0){
    printf("inet_pton error for %s\n",argv[1]);
    return 0;
}
if(connect(connfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0){
    printf("connect error: %s(errno: %d)\n",strerror(errno),errno);
    return 0;
}
```

下面则是设置了发送的超时时间和接收的超时时间为 2s。也就是说，如果接收一个包超过 2s 还没收到，就会断开连接，相关代码如下：

```
struct timeval stTimeValStruct;
stTimeValStruct.tv_sec = 2;
stTimeValStruct.tv_usec = 0;
if(setsockopt(connfd, SOL_SOCKET, SO_SNDTIMEO, &stTimeValStruct, sizeof(stTimeValStruct))){
    printf("setsockopt error: %s(errno: %d)\n", strerror(errno),errno);
    return 0;
}
if(setsockopt(connfd, SOL_SOCKET, SO_RCVTIMEO, &stTimeValStruct, sizeof(stTimeValStruct))){
    printf("setsockopt error: %s(errno: %d)\n", strerror(errno),errno);
    return 0;
}
```

如下是往服务端发 012345678 的字符串的相关代码：

```
ssize_t writeLen;
char sendMsg[10] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '\0' };
int count = 0;
writeLen = write(connfd, sendMsg, sizeof(sendMsg));
```

```

if (writeLen < 0) {
    printf("write error: %s(errno: %d)\n", strerror(errno), errno);
    close(connfd);
    return 0;
}
else{
    printf("write sucess, writelen :%d, sendMsg:%s\n",writeLen,sendMsg);
}

```

下面是接收来自服务端的回包的相关代码:

```

char readMsg[10]={0};
int readlen= read(connfd,readMsg,sizeof(readMsg));
if (readlen < 0) {
    printf("read error: %s(errno: %d)\n", strerror(errno), errno);
    close(connfd);
    return 0;
}
else{
    readMsg[9]= '\0';
    printf("read sucess, readlen :%d, readMsg:%s\n",readlen,readMsg);
}

```

在 server.cpp 中, 如下是在绑定端口和准备收包的相关代码:

```

if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ){
    printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
    return -1;
}
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(6666);
if( bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1){
    printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);
    return -1;
}
if( listen(listenfd, 10) == -1){
    printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);
    return -1;
}
printf("=====waiting for client's request=====\n");
if( (acceptfd = accept(listenfd, (struct sockaddr*)NULL, NULL)) == -1){
    printf("accept socket error: %s(errno: %d)",strerror(errno),errno);
}

```

下面是先收 100 个字节, 休眠 5s, 把收到的字符串中的第 2 个字节改成 9, 然后把新字符串发出去的相关代码:

```

char recvMsg[100];
ssize_t readLen = read(acceptfd, recvMsg, sizeof(recvMsg));
if (readLen < 0) {

```



```

printf("read error: %s(errno: %d)\n",strerror(errno),errno);
return -1;
}
recvMsg[9]='\0';
printf("readLen:%d, recvMsg:%s\n",readLen,recvMsg);
sleep(5);
recvMsg[1]='9';
ssize_t writeLen = write(acceptfd, recvMsg, sizeof(recvMsg));
printf("writeLen:%d, sendMsg:%s\n",writeLen,recvMsg);
if (writeLen < 0) {
    printf("writeLen error: %s(errno: %d)\n",strerror(errno),errno);
    return -1;
}

```

也就是说，客户端给服务端发了一个请求，服务端休眠 5s 之后再给客户端回包。但是因为客户端设置的收包超时时间只是 2s，所以客户端收包失败了，关闭了连接。

7. SO_RCVBUF 和 SO_SNDBUF

从写一个 TCP 套接字的 write 调用成功后返回，仅仅表示可以重新使用原来的应用进程缓冲区了，并不代表对端 TCP 或应用进程已接收到数据。对端 TCP 必须确认收到的数据，伴随来自对端的 ACK 的不断到达，本端 TCP 至此才能从套接字发送缓冲区中丢弃已确认的数据。TCP 必须为已发送的数据保留一个副本，直到它被对端确认为止。UDP 不保存应用进程数据的副本，因此无需一个真正的发送缓冲区，write 调用成功返回表示所写的的数据报或其所有分片已被加入数据链路层的输出队列。

对于 read 调用（套接字标志为阻塞），如果接收缓冲区中有 20 Byte 空间，请求读 100 Byte 的数据，就会返回 20。对于 write 调用（套接字标志为阻塞），如果请求写 100 Byte 的数据，而发送缓冲区中只有 20 Byte 的空闲位置，那么 write 会阻塞，直到把 100 Byte 的数据全部交给发送缓冲区才返回，如果 write 中得套接字标志为非阻塞，则直接返回 20，因此可以实现自己的 readn 和 writen 函数。

每个 TCP 套接字都有一个发送缓冲区和一个接收缓冲区，每个 UDP 套接字都有一个接收缓冲区。使用 SO_RCVBUF 和 SO_SNDBUF 这两个套接口选项可以改变默认缓冲区大小。

【例 6.3】发送缓冲区和接收缓冲区的展示。

client.cpp 的代码是：

```

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>

```

```

#define MAXLINE 4096

int main(int argc, char** argv){
    int connfd, n;
    char recvline[4096], sendline[4096];
    struct sockaddr_in servaddr;

    if( argc != 2){
        printf("usage: ./client <ipaddress>\n");
        return 0;
    }

    if( (connfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        printf("create socket error: %s(errno: %d)\n", strerror(errno), errno);
        return 0;
    }

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(6666);
    if( inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0){
        printf("inet_pton error for %s\n", argv[1]);
        return 0;
    }

    if(connect(connfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0){
        printf("connect error: %s(errno: %d)\n", strerror(errno), errno);
        return 0;
    }

    ssize_t writeLen;
    char sendMsg[246988] = {0};
    int count = 0;
    while (1){
        count++;
        if (count == 5) {
            return 0;
        }
        writeLen = write(connfd, sendMsg, sizeof(sendMsg));
        if (writeLen < 0) {
            printf("write failed\n");
            close(connfd);
            return 0;
        }
        else{
            printf("write success, writelen:%d\n", writeLen);
        }
    }
    close(connfd);
    return 0;
}

```

server.cpp 的代码是：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
#define MAXLINE 4096

int main(int argc, char** argv){
    int listenfd, acceptfd;
    struct sockaddr_in servaddr;
    if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ){
        printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
        return -1;
    }
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(6666);
    if( bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1 ){
        printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);
        return -1;
    }
    if( listen(listenfd, 10) == -1 ){
        printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);
        return -1;
    }
    printf("====waiting for client's request====\n");
    if( (acceptfd = accept(listenfd, (struct sockaddr*)NULL, NULL)) == -1 ){
        printf("accept socket error: %s(errno: %d)",strerror(errno),errno);
    }else{
        printf("accept succ\n");
        int rcvbuf_len;
        socklen_t len = sizeof(rcvbuf_len);
        if( getsockopt( acceptfd, SOL_SOCKET, SO_RCVBUF, (void *)&rcvbuf_len, &len
) < 0 ){
            perror("getsockopt: " );
        }
        printf("the recv buf len: %d\n" , rcvbuf_len );
    }
    char recvMsg[246988]={0};
    ssize_t totalLen = 0;
    while (1){
        sleep(1);
        ssize_t readLen = read(acceptfd, recvMsg, sizeof(recvMsg));
        printf("readLen:% ld\n",readLen);
        if (readLen < 0) {
```

```

        perror("read: ");
        return -1;
    }
    else if (readLen == 0) {
        printf("read finish: len = %ld\n", totalLen);
        close(acceptfd);
        return 0;
    }
    else{
        totalLen += readLen;
    }
}
close(acceptfd);
return 0;
}

```

makefile 的代码是:

```

all:server client
server:server.o
    g++ -g -o server server.o
client:client.o
    g++ -g -o client client.o
server.o:server.cpp
    g++ -g -c server.cpp
client.o:client.cpp
    g++ -g -c client.cpp
clean:all
    rm all

```

执行 make 命令后, 生成 server 和 client2 个可执行文件。分别打开两个终端窗口, 一个执行 ./server 命令, 一个执行 ./client 127.0.0.1 命令, 表示连上本机的 6666 端口, 执行 ./server 命令的要先执行。执行 ./client 127.0.0.1 命令后, 两个终端的展示结果如图 6-29 和图 6-30 所示:

```

[sharexu@linux 0603]$ ./client 127.0.0.1
write success, writelen:246988
write success, writelen:246988
write success, writelen:246988
[sharexu@linux 0603]$

```

图 6-29 例 6.3 客户端执行结果图

```

[sharexu@linux 0603]$ ./server
=====waiting for client's request=====
accept succ
the recv buf len: 87380
readLen: 65584
readLen: 65584
readLen: 147564
readLen: 147564
readLen: 246988
readLen: 246988
readLen: 67680
readLen: 0
read finish: len = 987952
[sharexu@linux 0603]$

```

图 6-30 例 6.3 服务端执行结果图

本例中, 服务端只负责收包, 所以程序 server.cpp 中打印了接收缓冲区的大小, 代码是:

```

if( getsockopt( acceptfd, SOL_SOCKET, SO_RCVBUF, (void *)&rcvbuf_len, &len ) < 0 ){

```

```

    perror("getsockopt: " );
}

```

注意，`perror` 也可以打印出相应的错误日志。

由程序的执行结果来看，接收缓冲区的大小是 87 380 Byte。

本例中，服务端并不是一直在收包，而是每收一次之后，就休眠一次，代码是：

```

while (1) {
    sleep(1);
    ssize_t readLen = read(acceptfd, recvMsg, sizeof(recvMsg));
    printf("readLen:% ld\n",readLen);
    if (readLen < 0) {
        perror("read: ");
        return -1;
    }
    else if (readLen == 0) {
        printf("read finish: len = % ld\n",totalLen);
        close(acceptfd);
        return 0;
    }
    else{
        totalLen += readLen;
    }
}

```

而客户端却一直不间断地发包，一共发 4 个包，每个包中发送 246 988 Byte。

```

ssize_t writeLen;
char sendMsg[246988] = {0};
int count = 0;
while (1){
    count++;
    if (count == 5) {
        return 0;
    }
    writeLen = write(connfd, sendMsg, sizeof(sendMsg));
    if (writeLen < 0) {
        printf("write failed\n");
        close(connfd);
        return 0;
    }
    else{
        printf("write sucess, writelen:%d\n",writeLen);
    }
}

```

也就是说，客户端发给了服务端数据，但服务端此时却还没有接收完毕就休眠了，等休眠完毕，会继续收包。最终，也把客户端发过来的 $246\,988 \times 4 = 987\,952$ Byte 全部接收完了。而这些未接收的数据，就是存储在接收缓冲区里的。

6.5 网络字节序与主机序

关于字节序，前面第1章已有所提过，这里再稍微做下分析。不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，称为主机序。最常见的有两种：

① Little Endian，将低序字节存储在起始地址；② Big Endian，将高序字节存储在起始地址。

Little Endian 把地址低位存储值的低位，地址高位存储值的高位。Little endian 是最符合人类思维的字节序，是因为从人的第一观感来说，低位值小，就应该放在内存地址小的地方，也即内存地址低位，反之，高位值就应该放在内存地址大的地方，也即内存地址高位。

Big Endian 把地址低位存储值的高位，地址高位存储值的低位。Big Endian 很直观，因为它不需考虑对应关系，只需要把内存地址从左到右按照由低到高的顺序写出，把值按照通常的高位到低位的顺序写出，两者对照，一个字节一个字节的填充进去即可。

用文字说明可能比较抽象，下面用图像加以说明。比如数字 0x12345678 在两种不同字节序 CPU 中的存储顺序如图 6-31 所示。

为什么要注意字节序的问题呢？当然，如果程序只在单机环境下运行，并且不和其他程序打交道，那么完全可以忽略字节序的存在；但是，如果程序要跟其他程序产生交互呢，此时就一定需要注意写字节序的问题。C/C++ 语言编写的程序里数据存储顺序是跟编译平台所在的 CPU 相关的，而 Java 编写的程序则唯一采用 Big Endian 方式来存储数据。

试想，如果你用 C/C++ 语言在 x86 平台下编写的程序跟别人的 Java 程序互通时会产生什么结果？就拿上面的 0x12345678 来说，将指向 0x12345678 的指针传给了 Java 程序，由于 JAVA 采取 Big Endian 方式存储数据，很自然会将你的数据翻译为 0x78563412。因此，在你的 C 程序传给 Java 程序之前有必要进行字节序的转换工作。

无独有偶，所有网络协议也都是采用 Big Endian 的方式来传输数据的。所以有时也会把 Big Endian 方式称之为网络字节序。当两台采用不同字节序的主机通信时，在发送数据之前都必须经过字节序的转换成为网络字节序后再进行传输。

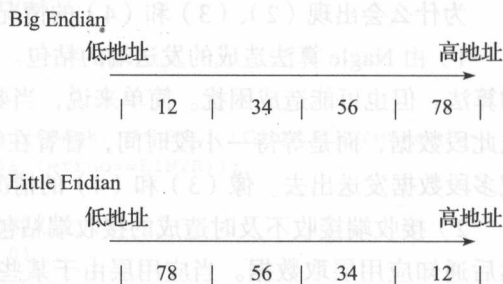


图 6-31 Big Endian 和 Little Endian 的存储示意图

6.6 封包和解包

TCP 是个“流”协议，所谓流，就是没有界限的一串数据。大家可以将其想象河里的流水，是连成一片的，其间是没有分界线的。但一般通信程序开发是需要定义一个个相互独立的数据包的，比如用于登录的数据包、用于注销的数据包等。由于 TCP “流”的特性以及网络状况，在进行数据传输时假设我们连续调用两次 send 分别发送两段数据 data1 和 data2，

在接收端有以下几种接收情况（当然不止这几种情况，这里只列出了有代表性的情况）。

（1）先接收到 data1，然后接收到 data2。

（2）先接收到 data1 的部分数据，然后接收到 data1 余下的部分以及 data2 的全部。

（3）先接收到了 data1 的全部数据和 data2 的部分数据，然后接收到了 data2 的余下的数据。

（4）一次性接收到了 data1 和 data2 的全部数据。

对于（1）这种情况正是我们需要的，不再做讨论。对于（2）、（3）和（4）的情况就是常说的“粘包”，就需要把接收到的数据进行拆包，拆成一个个独立的数据包；而为了拆包就必须在发送端进行封包。

对于 UDP 来说就不存在拆包的问题，因为 UDP 是个“数据包”协议，也就是两段数据间是有界限的，在接收端要么接收不到数据要么就是接收一段完整的数据，不会少接收也不会多接收。

为什么会出现（2）、（3）和（4）的情况呢，有以下几点原因。

1）由 Nagle 算法造成的发送端的粘包。前面有提到 Nagle 算法是一种改善网络传输效率的算法，但也可能造成困扰。简单来说，当要提交一段数据给 TCP 发送时，TCP 并不立刻发送此段数据，而是等待一小段时间，看看在等待期间是否还有要发送的数据，若有则会一次把多段数据发送出去。像（3）和（4）的情况就有可能由 Nagle 算法造成的。

2）接收端接收不及时造成的接收端粘包。TCP 会把接收到的数据存在自己的缓冲区中，然后通知应用层取数据。当应用层由于某些原因不能及时取出 TCP 的数据，就会造成 TCP 缓冲区中存放了多段数据。

“粘包”可发生在发送端也可发生在接收端。

最初遇到“粘包”的问题时，大家可能觉得可以在两次 send 之间调用 sleep 来休眠一小段时间，以此来解决。这个解决方法的缺点是显而易见的：使传输效率大大降低，而且也并不可靠。对数据包进行封包和拆包，就能解决这个问题。

封包就是给一段数据加上包头，这样一来数据包就分为包头和包体两部分内容了（以后讲过滤非法包时会加上“包尾”内容）。包头其实上是个大小固定的结构体，其中有个结构体成员变量表示包体的长度，这是个很重要的变量，其他的结构体成员可根据需要自己定义。根据固定的包头长度以及包头中含有的包体长度的变量值就能正确的拆分出一个完整的数据包。

利用底层的缓冲区来进行拆包时，由于 TCP 也维护了一个缓冲区，所以可以利用 TCP 的缓冲区来缓存发送的数据，这样一来就不需要为每一个连接分配一个缓冲区了，对于利用缓冲区来拆包，也就是循环不停地接收包头给出的数据，直到收够为止，这就是一个完整的 TCP 包。

为了解决“粘包”的问题，大家通常会在所发送的内容前，加上发送内容的长度，所以对方就会先收 4 Byte，解析获得接下来需要接收的长度，再进行收包。

1. 发送与接收一个字符串

【例 6.4】 发送与接收一个字符串。

server.cpp 的代码是:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/wait.h>
#include <string.h>
#include <errno.h>

int MyRecv( int iSock, char * pchBuf, size_t tCount){
    size_t tBytesRead=0;
    int iThisRead;
    while(tBytesRead < tCount){
        do{
            iThisRead = read(iSock, pchBuf, tCount-tBytesRead);
        } while((iThisRead<0) && (errno==EINTR));
        if(iThisRead < 0){
            return(iThisRead);
        }else if (iThisRead == 0)
            return(tBytesRead);
        tBytesRead += iThisRead;
        pchBuf += iThisRead;
    }
}

#define DEFAULT_PORT 6666

int main( int argc, char ** argv){
    int sockfd,acceptfd; /* 监听 socket: sock_fd, 数据传输 socket: acceptfd */
    struct sockaddr_in my_addr; /* 本机地址信息 */
    struct sockaddr_in their_addr; /* 客户地址信息 */
    unsigned int sin_size, myport=6666, lisnum=10;
    if ((sockfd = socket(AF_INET , SOCK_STREAM, 0)) == -1){
        perror("socket" );
        return -1;
    }
    printf("socket ok \n");
    my_addr.sin_family=AF_INET;
    my_addr.sin_port=htons(DEFAULT_PORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 0);
    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr )) == -1) {
        perror("bind" );
        return -2;
    }
}
```

```

printf("bind ok \n");
if (listen(sockfd, lisnum) == -1) {
    perror("listen" );
    return -3;
}
printf("listen ok \n");
char recvMsg[10];
sin_size = sizeof(my_addr);
acceptfd = accept(sockfd, (struct sockaddr *)&my_addr, &sin_size);
if (acceptfd < 0) {
    close(sockfd);
    printf("accept failed\n" );
    return -4;
}
ssize_t readLen = MyRecv(acceptfd, recvMsg, sizeof( int));
if (readLen < 0) {
    printf("read failed\n" );
    return -1;
}
int len=( int)ntohl(*( int*)recvMsg);
printf("len:%d\n", len);
readLen = MyRecv(acceptfd, recvMsg, len);
if (readLen < 0) {
    printf("read failed\n" );
    return -1;
}
recvMsg[len]='\0';
printf("recvMsg:%s\n" ,recvMsg);
close(acceptfd);
return 0;
}

```

client.cpp 的代码是:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet>
#include <errno.h>
int MySend( int iSock, char * pchBuf, size_t tLen){
    int iThisSend;
    unsigned int iSended=0;
    if(tLen == 0)
        return(0);
    while(iSended<tLen){
        do{
            iThisSend = send(iSock, pchBuf, tLen-iSended, 0);

```

```

    } while((iThisSend<0) && (errno==EINTR));
    if(iThisSend < 0){
        return(iSended);
    }
    iSended += iThisSend;
    pchBuf += iThisSend;
}
return(tLen);
}

#define DEFAULT_PORT 6666
int main( int argc, char * argv[]){
    int connfd = 0;
    int cLen = 0;
    struct sockaddr_in client;
    if(argc < 2){
        printf(" Usage: clientent [server IP address]\n");
        return -1;
    }
    client.sin_family = AF_INET;
    client.sin_port = htons(DEFAULT_PORT);
    client.sin_addr.s_addr = inet_addr(argv[1]);
    connfd = socket(AF_INET, SOCK_STREAM, 0);
    if(connfd < 0){
        printf("socket() failure!\n");
        return -1;
    }

    if(connect(connfd, (struct sockaddr*)&client, sizeof(client)) < 0){
        printf("connect() failure!\n");
        return -1;
    }

    ssize_t writeLen;
    char *sendMsg = "0123456789";
    int tLen=strlen(sendMsg);
    printf("tLen:%d\n",tLen);
    int iLen=0;
    char * pBuff= new char [100];
    *(int*)(pBuff+iLen)= htonl(tLen);
    iLen+=sizeof( int);
    memcpy(pBuff+iLen,sendMsg,tLen);
    iLen+=tLen;
    writeLen= MySend(connfd, pBuff, iLen);
    if (writeLen < 0) {
        printf("write failed\n");
        close(connfd);
        return 0;
    }
    else{
        printf("write sucess, writelen :%d, sendMsg:%s\n",writeLen,sendMsg);
    }
}

```



```

        close(connfd);
        return 0;
    }
}

```

makefile 的代码是：

```

all:server client
server:server.o
    g++ -g -o server server.o
client:client.o
    g++ -g -o client client.o
server.o:server.cpp
    g++ -g -c server.cpp
client.o:client.cpp
    g++ -g -c client.cpp
clean:all
    rm all

```

本例中，客户端给服务端发送了一个字符串，但是由于双方都不知道这个字符串会有多长，所以用发送的数据的前面 4 个字节表示这个字符串的大小。

```

ssize_t writeLen;
char *sendMsg = "0123456789";
int tLen=strlen(sendMsg);
printf("tLen:%d\n",tLen);
int iLen=0;
char * pBuff= new char [100];
*(int*)(pBuff+iLen)= htonl(tLen);
iLen+=sizeof( int);
memcpy(pBuff+iLen,sendMsg,tLen);
iLen+=tLen;

```

注意，一般要把字符串的长度转换成网络字节序，由于发送的内容是字符串，则无需转换网络字节序，直接加在后面即可。发送时，由于事先并不指定要发送的数据是多大，所以写了个函数，可以发送指定长度的数据。一次发送不完，可以接着发送，直到发送完指定长度为止，代码如下所示。

```

int MySend( int iSock, char * pchBuf, size_t tLen)
{
    int iThisSend;
    unsigned int iSended=0;
    if(tLen == 0)
        return(0);
    while(iSended<tLen){
        do{
            iThisSend = send(iSock, pchBuf, tLen-iSended, 0);
        } while((iThisSend<0) && (errno==EINTR));
        if(iThisSend < 0){
            return(iSended);
        }
    }
}

```

```

        iSended += iThisSend;
        pchBuf += iThisSend;
    }

    return(tLen);
}

```

而对于服务端，则需要先接收 4 个字节，并把它转换成主机序，这样才能知道接下来是接收多少字节的数据，如下所示：

```

ssize_t readLen = MyRecv(acceptfd, recvMsg, sizeof( int));
if (readLen < 0) {
    printf("read failed\n" );
    return -1;
}
int len=( int)ntohl(*( int*)recvMsg);
printf("len:%d\n",len);

```

由于事先并不知道会接收多少字节，所以也写了一个函数，用于循环接收，直到接收完指定数量为止，如下所示：

```

int MyRecv( int iSock, char * pchBuf, size_t tCount)
{
    size_t tBytesRead=0;
    int iThisRead;

    while(tBytesRead < tCount){
        do{
            iThisRead = read(iSock, pchBuf, tCount-tBytesRead);
        } while((iThisRead<0) && (errno==EINTR));
        if(iThisRead < 0){
            return(iThisRead);
        }
        else if (iThisRead == 0)
            return(tBytesRead);
        tBytesRead += iThisRead;
        pchBuf += iThisRead;
    }
}

```

接收完的数据，由于是字符串，所以无需再转换成主机序即可使用。注意，接收到的数据并没有结束符 '\0'，所以打印前需要加上结束符 '\0'。

```

recvMsg[len]='\0';
printf("recvMsg:%s\n" ,recvMsg);

```

程序执行结果如图 6-32 和图 6-33 所示。

那如果是想传输的内容是一个结构体，那么应该如果操作呢？接下来就要介绍发送与接收一个结构体的问题。

```
[sharexu@linux 0604]$ ./client 127.0.0.1
tLen:10
write success, writelen :14, sendMsg:0123456789
[sharexu@linux 0604]$
```

图 6-32 例 6.4 客户端执行结果图

```
[sharexu@linux 0604]$ ./server
socket ok
bind ok
listen ok
len:10
recvMsg:0123456789
[sharexu@linux 0604]$
```

图 6-33 例 6.4 服务端执行结果图

2. 发送与接收一个结构体

【例 6.5】传输一个结构体内容。

define.h 的代码是：

```
#pragma pack(1)
struct Header {
    int num ;           // 包 id
    int index ;         // 学生编号
};
struct PkgContent {
    char sex ;          // 性别
    int score ;         // 分数
    char address [100]; // 地址
    int age;
};
struct Pkg {
    Header head;
    PkgContent content ;
};

int MySend( int iSock, char * pchBuf, size_t tLen){
    int iThisSend;
    unsigned int iSended=0;
    if(tLen == 0)
        return 0;
    while(iSended<tLen){
        do{
            iThisSend = send(iSock, pchBuf, tLen-iSended, 0);
        } while((iThisSend<0) && (errno==EINTR));
        if(iThisSend < 0){
            return(iSended);
        }
        iSended += iThisSend;
        pchBuf += iThisSend;
    }
    return(tLen);
}

int MyRecv( int iSock, char * pchBuf, size_t tCount){
    size_t tBytesRead=0;
    int iThisRead;
```

```

while(tBytesRead < tCount){
    do{
        iThisRead = read(iSock, pchBuf, tCount-tBytesRead);
    } while((iThisRead<0) && (errno==EINTR));
    if(iThisRead < 0){
        return(iThisRead);
    }
    else if (iThisRead == 0)
        return(tBytesRead);
    tBytesRead += iThisRead;
    pchBuf += iThisRead;
}
}

```

server.cpp 的代码是:

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/wait.h>
#include <string.h>
#include <errno.h>
#include "define.h"
#define DEFAULT_PORT 6666

int main( int argc, char ** argv){
    int sockfd,acceptfd; /* 监听 socket: sockfd, 数据传输 socket: acceptfd */
    struct sockaddr_in my_addr; /* 本机地址信息 */
    struct sockaddr_in their_addr; /* 客户地址信息 */
    unsigned int sin_size, myport=6666, lisnum=10;
    if ((sockfd = socket(AF_INET , SOCK_STREAM, 0)) == -1) {
        perror("socket" );
        return -1;
    }
    printf("socket ok \n");
    my_addr.sin_family=AF_INET;
    my_addr.sin_port=htons(DEFAULT_PORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 0);
    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
        perror("bind" );
        return -2;
    }
    printf("bind ok \n");
    if (listen(sockfd, lisnum) == -1) {
        perror("listen" );
        return -3;
    }
}

```

```

}
printf("listen ok \n");
char recvMsg[1000];
sin_size = sizeof(my_addr);
acceptfd = accept(sockfd, (struct sockaddr *)&my_addr, &sin_size);
if (acceptfd < 0) {
    close(sockfd);
    printf("accept failed\n");
    return -4;
}
ssize_t readLen = MyRecv(acceptfd, recvMsg, sizeof(int));
if (readLen < 0) {
    printf("read failed\n");
    return -1;
}
int len = (int)ntohl(*(int*)recvMsg);
printf("len:%d\n", len);
readLen = MyRecv(acceptfd, recvMsg, len);
if (readLen < 0) {
    printf("read failed\n");
    return -1;
}
char * pBuff = recvMsg;
Pkg RecvPkg;
int iLen = 0;
memcpy(&RecvPkg.head.num, pBuff + iLen, sizeof(int));
iLen += sizeof(int);
RecvPkg.head.num = ntohl(RecvPkg.head.num);
printf("RecvPkg.head.num:%d\n", RecvPkg.head.num);
memcpy(&RecvPkg.head.index, pBuff + iLen, sizeof(int));
iLen += sizeof(int);
RecvPkg.head.index = ntohl(RecvPkg.head.index);
printf("RecvPkg.head.index:%d\n", RecvPkg.head.index);
memcpy(&RecvPkg.content.sex, pBuff + iLen, sizeof(char));
iLen += sizeof(char);
printf("RecvPkg.content.sex:%c\n", RecvPkg.content.sex);
memcpy(&RecvPkg.content.score, pBuff + iLen, sizeof(int));
iLen += sizeof(int);
RecvPkg.content.score = ntohl(RecvPkg.content.score);
printf("RecvPkg.content.score:%d\n", RecvPkg.content.score);
memcpy(&RecvPkg.content.address, pBuff + iLen, sizeof(RecvPkg.content.address));
iLen += sizeof(RecvPkg.content.address);
printf("RecvPkg.content.address:%s\n", RecvPkg.content.address);
memcpy(&RecvPkg.content.age, pBuff + iLen, sizeof(int));
iLen += sizeof(int);
RecvPkg.content.age = ntohl(RecvPkg.content.age);
printf("RecvPkg.content.age:%d\n", RecvPkg.content.age);
close(acceptfd);
return 0;
}

```


client.cpp 的代码是:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include "define.h"
#define DEFAULT_PORT 6666

int main( int argc, char * argv[]){
    int connfd = 0;
    int cLen = 0;
    struct sockaddr_in client;
    if(argc < 2){
        printf(" Usage: clientent [server IP address]\n");
        return -1;
    }
    client.sin_family = AF_INET;
    client.sin_port = htons(DEFAULT_PORT);
    client.sin_addr.s_addr = inet_addr(argv[1]);
    connfd = socket(AF_INET, SOCK_STREAM, 0);
    if(connfd < 0){
        printf("socket() failure!\n" );
        return -1;
    }
    if(connect(connfd, (struct sockaddr*)&client, sizeof(client)) < 0){
        printf("connect() failure!\n" );
        return -1;
    }
    Pkg mypkg;
    mypkg.head.num=1;
    mypkg.head.index=10001;
    mypkg.content.sex='m';
    mypkg.content.score=90;
    char * temp="guangzhou and shanghai";
    strncpy(mypkg.content.address,temp,sizeof(mypkg.content.address));
    mypkg.content.age=18;
    ssize_t writeLen;
    int tLen=sizeof(mypkg);
    printf("tLen:%d\n",tLen);
    int iLen=0;
    char * pBuff= new char [1000];
    *(int*)(pBuff+iLen)= htonl(tLen);
    iLen+=sizeof( int);
    *(int*)(pBuff+iLen)= htonl(mypkg.head.num);
    iLen+=sizeof( int);
    *(int*)(pBuff+iLen)= htonl(mypkg.head.index);
```

```

iLen+=sizeof( int);
memcpy(pBuff+iLen,&mypkg.content.sex,sizeof( char));
iLen+=sizeof( char);
*(int*)(pBuff+iLen)= htonl(mypkg.content.score);
iLen+=sizeof( int);
memcpy(pBuff+iLen,mypkg.content.address,sizeof(mypkg.content.address));
iLen+=(sizeof(mypkg.content.address));
*(int*)(pBuff+iLen)= htonl(mypkg.content.age);
iLen+=sizeof( int);
writeLen= MySend(connfd, pBuff, iLen);
if (writeLen < 0) {
    printf("write failed\n" );
    close(connfd);
    return 0;
}
else{
    printf("write sucess, writelen :%d, iLen:%d, pBuff: %s\n",writeLen,iLen,pBuff);
}
close(connfd);
return 0;
}

```

makefile 的代码是:

```

all:server client
server:server.o
    g++ -g -o server server.o
client:client.o
    g++ -g -o client client.o
server.o:server.cpp define.h
    g++ -g -c server.cpp
client.o:client.cpp define.h
    g++ -g -c client.cpp
clean:all
    rm all

```

执行 make 命令后, 生成 server 和 client 2 个可执行文件。分别打开两个终端窗口, 一个执行 ./server 命令, 一个执行 ./client 127.0.0.1 命令, 表示连上本机的 6666 端口, 并且执行 ./server 命令的要先执行。执行结果如图 6-34 和图 6-35 所示。

例 6.5 中, 由于 server 和 client 都将处理同样的结构体内容, 所以把即将传输的结构体内容定义在 define.h 文件中, 有包头和包体, 代码如下所示。

```

#pragma pack(1)
struct Header {

```

```

[sharexu@linux 0605]$ ./client 127.0.0.1
tlen:117
write sucess, writelen :121, iLen:121, pBuff:

```

图 6-34 例 6.5 客户端执行结果图

```

[sharexu@linux 0605]$ ./server
socket ok
bind ok
listen ok
len:117
RecvPkg.head.num:1
RecvPkg.head.index:10001
RecvPkg.content.sex:m
RecvPkg.content.score:90
RecvPkg.content.address:guangzhou and shanghai
RecvPkg.content.age:18


```

图 6-35 例 6.5 服务器端执行结果图

```

        int num ;           // 包 id
        int index ;        // 学生编号
    };
    struct PkgContent {
        char sex ;          // 性别
        int score ;         // 分数
        char address [100]; // 地址
        int age;
    };
    struct Pkg {
        Header head;
        PkgContent content ;
    };

```

 **注意** 这里有一行代码：“#pragma pack (1)”，这是按照单字节对齐的意思。如果不加此句，则 `sizeof (Pkg)` 的值为 120；而如果按照单字节对齐后，`sizeof (Pkg)` 则为 117。这是为了适应不同的机器，保证服务器和客户端都能在对应位置上取到对应的值。

`client.cpp` 中对结构体 `mypkg` 赋值后，就开始逐个元素进行网络字节序的转换。虽然传输的结构体有 `char` 数组，计算长度时用 `sizeof (结构体)` 即可，方便 `server` 收到包时也这样逐个解析，代码如下所示。

```
int tLen=sizeof(mypkg);
```

下面是分别对结构体的每个元素进行网络字节序的转换，然后复制到一个 `char` 数组里。这里面有 `char` 类型的复制，也有 `char` 数组类型的复制，代码如下所示。

```

*(int*)(pBuff+iLen)= htonl(tLen);
iLen+=sizeof( int);
*(int*)(pBuff+iLen)= htonl(mypkg.head.num);
iLen+=sizeof( int);
*(int*)(pBuff+iLen)= htonl(mypkg.head.index);
iLen+=sizeof( int);
memcpy(pBuff+iLen,&mypkg.content.sex,sizeof( char));
iLen+=sizeof( char);
*(int*)(pBuff+iLen)= htonl(mypkg.content.score);
iLen+=sizeof( int);
memcpy(pBuff+iLen,mypkg.content.address,sizeof(mypkg.content.address));
iLen+=(sizeof(mypkg.content.address));
*(int*)(pBuff+iLen)= htonl(mypkg.content.age);
iLen+=sizeof( int);

```

然后就是发送数据，代码如下所示。

```

writeLen= MySend(connfd, pBuff, iLen);
if (writeLen < 0) {

```

```

printf("write failed\n" );
close(connfd);
return 0;
}

```

server.cpp 中, 先收 4 个字节的包, 然后再将其转换为主机序, 得到接下来该接收的长度, 代码如下所示。

```

ssize_t readLen = MyRecv(acceptfd, recvMsg, sizeof(int));
if (readLen < 0) {
    printf("read failed\n" );
    return -1;
}
int len=(int)ntohl(*(int*)recvMsg);
printf("len:%d\n",len);
readLen = MyRecv(acceptfd, recvMsg, len);
if (readLen < 0) {
    printf("read failed\n" );
    return -1;
}

```

收到包后, 就是把各个元素逐个解析出来, 代码如下所示。

```

memcpy(&RecvPkg.head.num , pBuff + iLen, sizeof( int));
iLen += sizeof(int);
RecvPkg. head. num = ntohl(RecvPkg.head.num);
printf("RecvPkg.head.num:%d\n" ,RecvPkg.head.num);
memcpy(&RecvPkg.head.index , pBuff + iLen, sizeof( int));
iLen += sizeof(int);
RecvPkg. head. index = ntohl(RecvPkg.head.index);
printf("RecvPkg.head.index:%d\n" ,RecvPkg.head.index);
memcpy(&RecvPkg.content.sex , pBuff + iLen, sizeof( char));
iLen += sizeof(char);
printf("RecvPkg.content.sex:%c\n" ,RecvPkg.content.sex);
memcpy(&RecvPkg.content.score , pBuff + iLen, sizeof( int));
iLen += sizeof(int);
RecvPkg. content.score = ntohl(RecvPkg. content.score );
printf("RecvPkg.content.score:%d\n" ,RecvPkg.content.score);
memcpy(&RecvPkg.content.address, pBuff + iLen, sizeof(RecvPkg.content.address ));
iLen += sizeof(RecvPkg.content.address);
printf("RecvPkg.content.address:%s\n" ,RecvPkg.content.address);
memcpy(&RecvPkg.content.age , pBuff + iLen, sizeof( int));
iLen += sizeof(int);
RecvPkg.content.age = ntohl(RecvPkg.content.age );
printf("RecvPkg.content.age:%d\n" ,RecvPkg.content.age);

```

memcpy 是 C 和 C++ 使用的内存拷贝函数, 其功能是从源 src 所指的内存地址的起始位置开始拷贝 n 个字节到目标 dest 所指的内存地址的起始位置中。函数原型是这样的:

```

void *memcpy(void *dest, const void *src, size_t n);

```

当然，现在也不用这么麻烦地逐个解析了，有 `protobuf` 等自动生成解析数据的函数，这部分内容在后面的章节将会详细讲解。

6.7 本章小结

本章介绍了 TCP 协议及网络编程 API，带大家实现了一个 TCP server，还介绍了 TCP 协议选项、网络字节序和主机序，以及如何封包与解包。熟练使用这些知识，足以让你轻松地写一个 server 了。

不过，本章中实现的 server，在同一时刻只能接收一个包，对于互联网的海量请求，这是远远不够的，所以接下来的第 7 章将继续探索如何写出高并发量的 server。

网络 IO 模型

IO (Input/Output, 输入/输出) 是计算机体系中重要的一部分。IO 类外设有打印机、键盘、复印机等; 存储类型的设备则有硬盘、磁盘、U 盘等; 通信设备有网卡、路由器等。不同的 IO 设备有着不同的特点: 数据率不一样、传送单位不一样、数据表示不一样, 等等。所以, 很难实现一种统一的输入/输出方法。

IO 有两种操作, 同步 IO 和异步 IO。同步 IO 指的是, 必须等待 IO 操作完成后, 控制权才返回给用户进程。异步 IO 指的是, 无须等待 IO 操作完成, 就将控制权返回给用户进程。

网络中的 IO, 由于不同的 IO 设备有着不同的特点, 网络通信中往往需要等待。常见的有以下 4 种情况。

- (1) 输入操作: 等待数据到达套接字接收缓冲区。
- (2) 输出操作: 等待套接字发送缓冲区有足够的空间容纳将要发送的数据。
- (3) 服务器接收连接请求: 等待新的客户端连接请求的到来。
- (4) 客户端发送连接请求: 等待服务器回送客户的发起的 SYN 所对应的 ACK。

当一个网络 IO (假设是 read) 发生时, 它会涉及两个系统对象, 一个是调用这个 IO 的进程, 另一个是系统内核。当一个 read 操作发生时, 它会经历两个阶段: ①等待数据准备; ②将数据从内核拷贝到进程中。

本章主要讲网络通信中的 IO 操作。

7.1 4 种网络 IO 模型

为了解决网络 IO 中的问题, 学者们提出了 4 种网络 IO 模型: ①阻塞 IO 模型; ②非阻

塞IO模型；③多路IO复用模型；④异步IO模型。

下面对这4种模型进行具体讲解。

1. 阻塞IO模型

在Linux中，默认情况下所有的socket都是阻塞的，一个典型的读操作流程如图7-1所示。

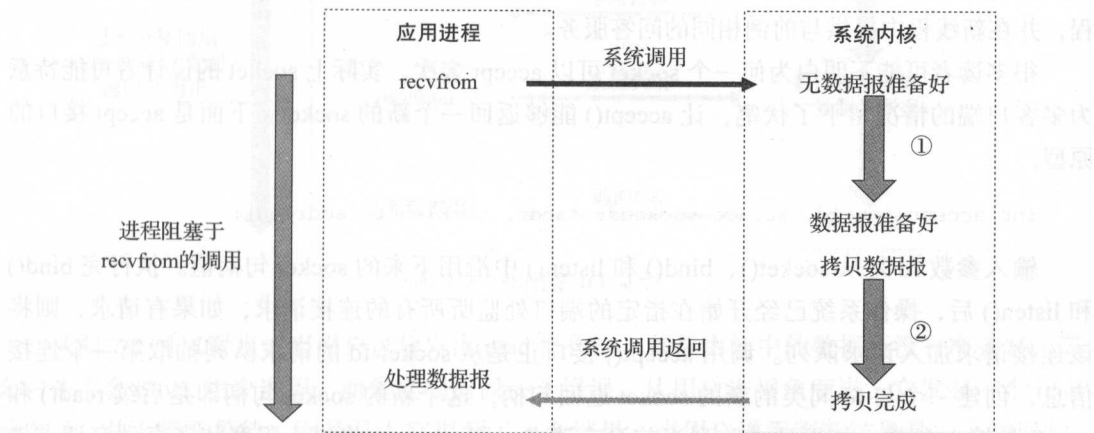


图 7-1 阻塞IO模型

阻塞和非阻塞的概念描述的是用户线程调用内核IO操作的方式：阻塞是指IO操作需要彻底完成后才返回到用户空间；而非阻塞是指IO操作被调用后立即返回给用户一个状态值，不需要等到IO操作彻底完成。

当应用进程调用了recvfrom这个系统调用后，系统内核就开始了IO的第一个阶段：准备数据。对于网络IO来说，很多时候数据在一开始还没到达时（比如还没有收到一个完整的TCP包），系统内核就要等待足够的数据到来。而在用户进程这边，整个进程会被阻塞。当系统内核一直等到数据准备好了，它就会将数据从系统内核中拷贝到用户内存中，然后系统内核返回结果，用户进程才解除阻塞的状态，重新运行起来。所以，阻塞IO模型的特点就是在IO执行的两个阶段（等待数据和拷贝数据）都被阻塞了。

大部分的socket接口都是阻塞型的。所谓阻塞型接口是指系统调用时（一般是IO接口）却不返回调用结果，并让当前线程一直处于阻塞状态，只有当该系统调用获得结果或者超时出错时才返回结果。实际上，除非特别指定，几乎所有的IO接口（包括socket接口）都是阻塞型的。这给网络编程带来了一个很大的问题，如在调用send()的同时，线程处于阻塞状态，则在此期间，线程将无法执行任何运算或响应任何网络请求。

一个简单的改进方案是在服务器端使用多线程（或多进程）。多线程（或多进程）的目的是让每个连接都拥有独立的线程（或进程），这样任何一个连接的阻塞都不会影响其他的连接。具体使用多进程还是多线程，并没有一个特定的模式。传统意义上，进程的开销要远远

大于线程，所以如果需要同时为较多的客户端提供服务，则不推荐使用多进程；如果单个服务执行体需要消耗较多的 CPU 资源，例如需要进行大规模或长时间的数据运算或文件访问，则推荐使用较为安全的进程。通常，使用 `pthread_create()` 创建新线程，使用 `fork()` 创建新进程。（多线程和多进程将在后续章节详细学习。）

假设对前面第 6 章举例中的服务器客户端模型提出更高的要求，即让服务器同时为多个客户端提供一问一答的服务。主线程持续等待客户端的连接请求，如果有连接，则创建新线程，并在新线程中提供与前例相同的问答服务。

很多读者可能不明白为何一个 `socket` 可以 `accept` 多次。实际上 `socket` 的设计者可能特意为多客户端的情况留下了伏笔，让 `accept()` 能够返回一个新的 `socket`。下面是 `accept` 接口的原型：

```
int accept(int fd, struct sockaddr *addr, socklen_t *addrlen);
```

输入参数 `fd` 是从 `socket()`、`bind()` 和 `listen()` 中沿用下来的 `socket` 句柄值。执行完 `bind()` 和 `listen()` 后，操作系统已经开始在指定的端口处监听所有的连接请求，如果有请求，则将该连接请求加入请求队列。调用 `accept()` 接口正是从 `socket fd` 的请求队列抽取第一个连接信息，创建一个与 `fd` 同类的新的 `socket` 返回句柄，这个新的 `socket` 句柄即是后续 `read()` 和 `recv()` 的输入参数。如果请求队列当前没有请求，则 `accept()` 将进入阻塞状态直到有请求进入队列。

上述多线程的服务器模型似乎完美地解决了为多个客户机提供问答服务的要求，但其实并不尽然。如果要同时响应成百上千路的连接请求，则无论多线程还是多进程都会严重占据系统资源，降低系统对外界响应的效率，而线程与进程本身也更容易进入假死状态。

很多程序员可能会考虑使用“线程池”或“连接池”。“线程池”旨在降低创建和销毁线程的频率，使其维持一定合理数量的线程，并让空闲的线程重新承担新的执行任务。“连接池”是指维持连接的缓存池，尽量重用已有的连接，降低创建和关闭连接的频率。这两种技术都可以很好地降低系统开销，都被广泛应用于很多大型系统。但是，“线程池”和“连接池”技术也只是在一定程度上缓解了频繁调用 IO 接口带来的资源占用。而且，所谓“池”始终有其上限，当请求大大超过上限时，“池”构成的系统对外界的响应并不比没有“池”的时候效果好多少。所以使用“池”必须考虑其面临的响应规模，并根据响应规模调整“池”的大小。

现实生活所面临的可能是同时出现的上千甚至上万次的客户端请求，“线程池”或“连接池”或许可以缓解部分压力，但是不能解决所有问题。总之，多线程模型可以方便高效的解决小规模的服务请求，但面对大规模的服务请求，多线程模型也会遇到瓶颈，可以用非阻塞模型来尝试解决这个问题。

2. 非阻塞 IO 模型

在 Linux 下，可以通过设置 `socket` 使 IO 变为非阻塞状态。当对一个非阻塞的 `socket` 执

行 read 操作时，流程如图 7-2 所示。

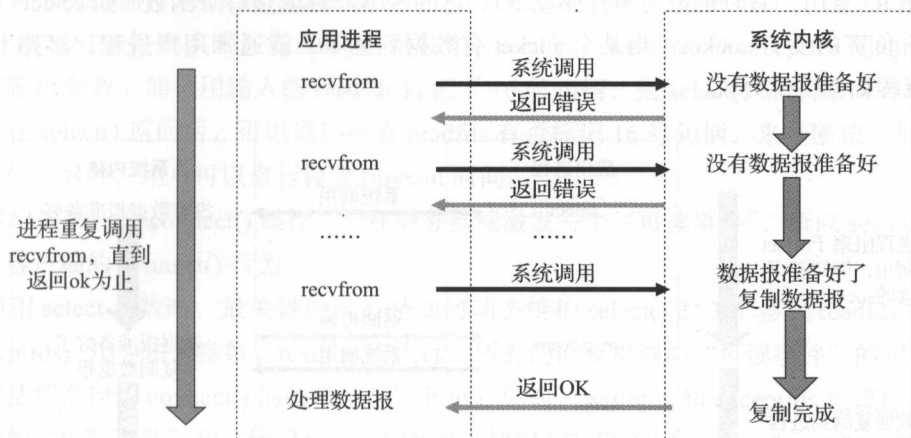


图 7-2 非阻塞 IO 模型

从图 7-2 可以看出，当用户进程发出 read 操作时，如果内核中的数据还没有准备好，那么它并不会 block 用户进程，而是立刻返回一个错误。从用户进程角度讲，它发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。当用户进程判断结果是一个错误时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦内核中的数据准备好了，并且又再次收到了用户进程的系统调用，那么它马上就数据复制到了用户内存中，然后返回正确的返回值。

所以，在非阻塞式 IO 中，用户进程其实需要不断地主动询问 kernel 数据是否准备好。非阻塞的接口相比于阻塞型接口的显著差异在于被调用之后立即返回。使用如下的函数可以将某句柄 fd 设为非阻塞状态：

```
fcntl( fd, F_SETFL, O_NONBLOCK );
```

在非阻塞状态下，recv() 接口在被调用后立即返回，返回值代表了不同的含义，如下所述。

- (1) recv() 返回值大于 0，表示接收数据完毕，返回值即是接收到的字节数。
- (2) recv() 返回 0，表示连接已经正常断开。
- (3) recv() 返回 -1，且 errno 等于 EAGAIN，表示 recv 操作还没执行完成。
- (4) recv() 返回 -1，且 errno 不等于 EAGAIN，表示 recv 操作遇到系统错误 errno。

可以看到服务器线程可以通过循环调用 recv() 接口，可以在单个线程内实现对所有连接的数据接收工作。但是上述模型绝不被推荐，因为循环调用 recv() 将大幅度占用 CPU 使用率；此外，在这个方案中 recv() 更多的是起到检测“操作是否完成”的作用，实际操作系统提供了更为高效的检测“操作是否完成”作用的接口，例如 select() 多路复用模式，可以一次检测多个连接是否活跃。

3. 多路 IO 复用模型

多路 IO 复用，有时也称为事件驱动 IO。它的基本原理就是有个函数（如 `select`）会不断地轮询所负责的所有 `socket`，当某个 `socket` 有数据到达了，就通知用户进程，多路 IO 复用模型的流程如图 7-3 所示。

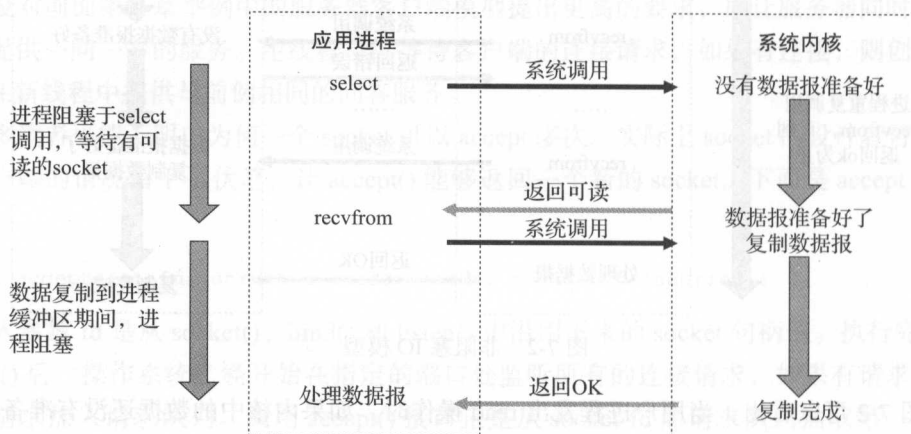


图 7-3 多路 IO 复用模型的流程图

当用户进程调用了 `select`，那么整个进程会被阻塞，而同时，内核会“监视”所有 `select` 负责的 `socket`，当任何一个 `socket` 中的数据准备好了，`select` 就会返回。这个时候用户进程再调用 `read` 操作，将数据从内核拷贝到用户进程。

这个模型和阻塞 IO 的模型其实并没有太大的不同，事实上还更差一些。因为这里需要使用两个系统调用（`select` 和 `recvfrom`），而阻塞 IO 只调用了—个系统调用（`recvfrom`）。但是，用 `select` 的优势在于它可以同时处理多个连接。所以，如果处理的连接数不是很高的话，使用 `select/epoll` 的 Web server 不一定比使用多线程的阻塞 IO 的 Web server 性能更好，可能延迟还更大；`select/epoll` 的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。

在多路复用 IO 模型中，对于每一个 `socket`，一般都设置成为非阻塞的，但是，如图 7-3 所示，整个用户的进程其实是一直被阻塞的。只不过进程是被 `select` 这个函数阻塞，而不是被 `socket` IO 阻塞。因此使用 `select()` 的效果与非阻塞 IO 类似。

大部分 UNIX/Linux 都支持 `select` 函数，该函数用于探测多个文件句柄的状态变化。下面给出 `select` 接口的原型：

```

FD_ZERO(int fd, fd_set* fds) ;
FD_SET(int fd, fd_set* fds) ;
FD_ISSET(int fd, fd_set* fds) ;
FD_CLR(int fd, fd_set* fds) ;
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout) ;
  
```

这里, `fd_set` 类型可以简单理解为按 bit 位标记句柄的队列, 例如要在某 `fd_set` 中标记一个值为 16 的句柄, 则该 `fd_set` 的第 16 个 bit 位被标记为 1。具体的置位、验证可使用 `FD_SET`、`FD_ISSET` 等宏实现。在 `select()` 函数中, `readfds`、`writefds` 和 `exceptfds` 同时作为输入参数和输出参数。如果用输入的 `readfds` 标记了 16 号句柄, 则 `select()` 将检测 16 号句柄是否可读。在 `select()` 返回后, 可以通过检查 `readfds` 有否标记 16 号句柄, 来判断该“可读事件”是否发生。另外, 用户可以自行设置 timeout 时间。

客户端的一个 `connect()` 操作, 将在服务器端激发一个“可读事件”, 所以 `select()` 也能检测来自客户端的 `connect()` 行为。

使用 `select` 函数时, 最关键的地方是如何动态维护 `select()` 的 3 个参数 `readfds`、`writefds` 和 `exceptfds`。作为输入参数, `readfds` 应该标记所有的需要检测的“可读事件”的句柄, 其中永远包括那个检测 `connect()` 的那个“母”句柄; 同时, `writefds` 和 `exceptfds` 应该标记所有需要检测的“可写事件”和“错误事件”的句柄 (使用 `FD_SET()` 标记)。

作为输出参数, `readfds`、`writefds` 和 `exceptfds` 中保存了 `select()` 捕捉到的所有事件的句柄值。程序员需要检查所有的标记位 (使用 `FD_ISSET()` 检查), 以确定到底哪些句柄发生了事件。

如果 `select()` 发现某句柄捕捉到了“可读事件”, 服务器程序应及时做 `recv()` 操作, 并根据接收到的数据准备好待发送数据, 并将对应的句柄值加入 `writefds`, 准备下一次的“可写事件”的 `select()` 检测。同样, 如果 `select()` 发现某句柄捕捉到“可写事件”, 则程序应及时做 `send()` 操作, 并准备好下一次的“可读事件”检测准备。图 7-4 描述了多路 IO 复用模型中的一个执行周期。

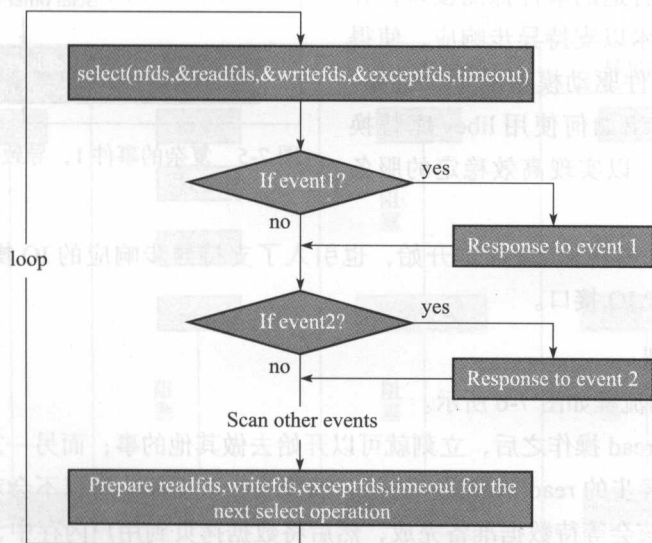


图 7-4 多路 IO 复用模型的一个执行周期

这种模型的特征在于每一个执行周期都会探测一次或一组事件，一个特定的事件会触发某个特定的响应，这里可以将这种模型归类为“事件驱动模型”。

相比其他模型，使用 `select()` 的事件驱动模型只用单线程（进程）执行，占用资源少，不消耗太多 CPU 资源，同时能够为多客户端提供服务。如果试图建立一个简单的事件驱动的服务器程序，这个模型有一定的参考价值。

但这个模型依旧有着很多问题。首先 `select()` 接口并不是实现“事件驱动”的最好选择。因为当需要探测的句柄值较大时，`select()` 接口本身需要消耗大量时间去轮询各个句柄。很多操作系统提供了更为高效的接口，如 Linux 提供了 `epoll`，BSD 提供了 `kqueue`，Solaris 提供了 `/dev/poll` 等。如果需要进行更高效的服务器程序，则更推荐使用类似 `epoll` 这样的接口。遗憾的是，不同的操作系统特供的 `epoll` 接口有很大差异，所以使用类似于 `epoll` 的接口实现具有较好跨平台能力的服务器会比较困难。

其次，该模型将事件探测和事件响应夹杂在一起，一旦事件响应的执行体过于庞大，则对整个模型是灾难性的。如图 7-5 所示，庞大的执行体 1 将直接导致响应事件 2 的执行体迟迟得不到执行，并在很大程度上降低了事件检测的及时性。

幸运的是，有很多高效的事件驱动库可以屏蔽上述难题，常见的事件驱动库有 `libevent` 库、`libev` 库等。这些库会根据操作系统的特点选择最合适的事件探测接口，并且加入了相应的技术以支持异步响应，使得这些库成为构建事件驱动模型的不二选择。下面的第 8 章将介绍如何使用 `libev` 库替换 `select` 或 `epoll` 接口，以实现高效稳定的服务器模型。

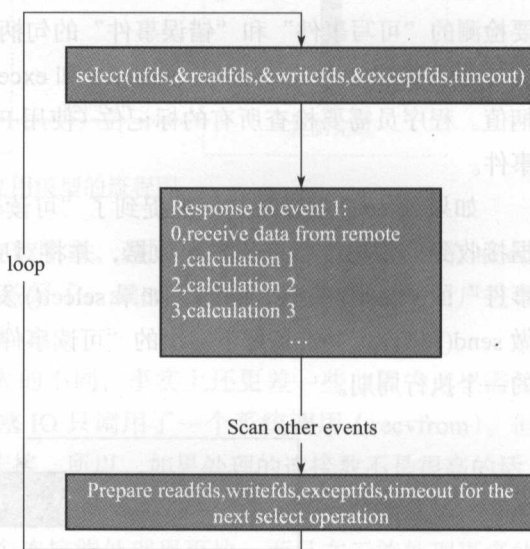


图 7-5 复杂的事件 1，导致事件 2 迟迟未执行

实际上，Linux 内核从 2.6 版本开始，也引入了支持异步响应的 IO 操作，如 `aio_read`、`aio_write`，就是异步 IO 接口。

4. 异步 IO 模型

异步 IO 模型的流程如图 7-6 所示。

用户进程发起 `read` 操作之后，立刻就可以开始去做其他的事；而另一方面，从内核的角度，当它收到一个异步的 `read` 请求操作之后，首先会立刻返回，所以不会对用户进程产生任何阻塞。然后，内核会等待数据准备完成，然后将数据拷贝到用户内存中，当这一切都完成之后，内核会给用户进程发送一个信号，返回 `read` 操作已完成的信息。

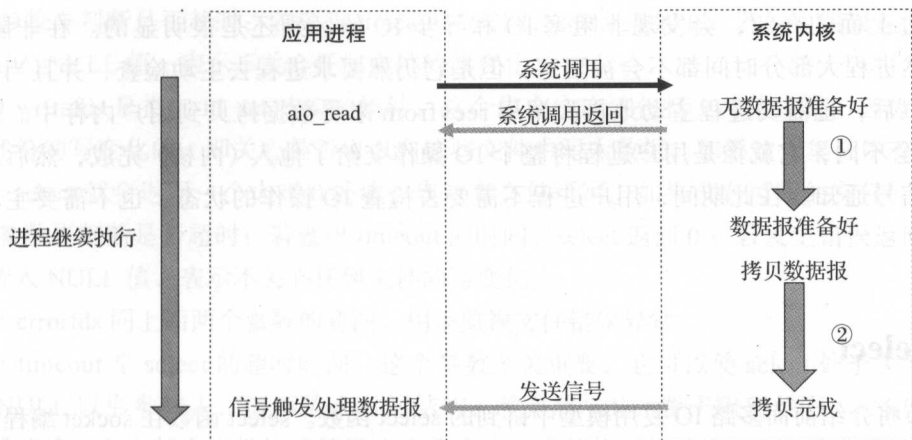


图 7-6 异步 IO 模型流程图

调用阻塞 IO 会一直阻塞住对应的进程直到操作完成，而非阻塞 IO 在内核还在准备数据的情况下会立刻返回。两者的区别就在于同步 IO 进行 IO 操作时会阻塞进程。按照这个定义，之前所述的阻塞 IO、非阻塞 IO 及多路 IO 复用都属于同步 IO。实际上，真实的 IO 操作，就是例子中的 `recvfrom` 这个系统调用。非阻塞 IO 在执行 `recvfrom` 这个系统调用的时候，如果内核的数据没有准备好，这时候不会阻塞进程。但是当内核中数据准备好时，`recvfrom` 会将数据从内核拷贝到用户内存中，这个时候进程则被阻塞。而异步 IO 则不一样，当进程发起 IO 操作之后，就直接返回，直到内核发送一个信号，告诉进程 IO 已完成，则在这整个过程中，进程完全没有被阻塞。

各个 IO 模型的比较如图 7-7 所示。

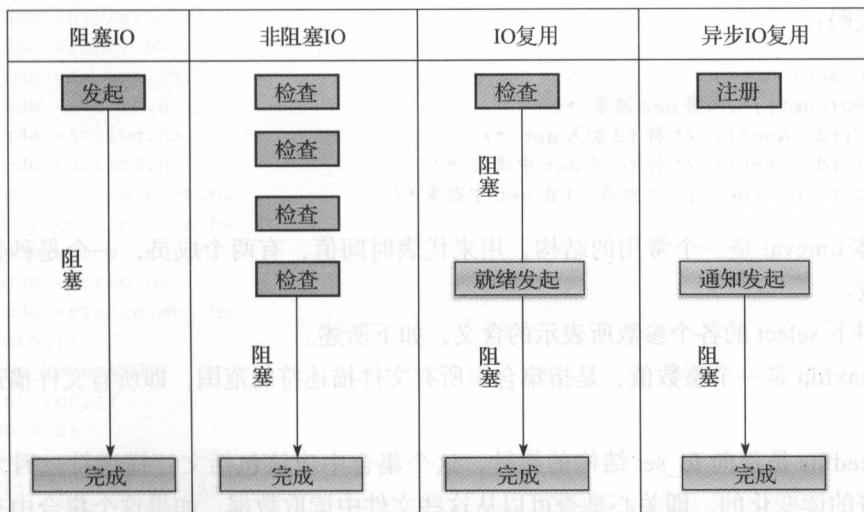


图 7-7 各种 IO 模型的比较

经过上面的介绍，会发现非阻塞 IO 和异步 IO 的区别还是很明显的。在非阻塞 IO 中，虽然进程大部分时间都不会被阻塞，但是它仍然要求进程去主动检查，并且当数据准备完成以后，也需要进程主动地再次调用 `recvfrom` 来将数据拷贝到用户内存中。而异步 IO 则完全不同，它就像是用户进程将整个 IO 操作交给了他人（内核）完成，然后内核做完后发信号通知。在此期间，用户进程不需要去检查 IO 操作的状态，也不需要主动地拷贝数据。

7.2 select

本节将介绍前面多路 IO 复用模型中讲到的 `select` 函数。`select` 函数在 socket 编程中还是比较重要的，可是很多初学 socket 的人可能并不爱用 `select` 写程序，而习惯写诸如 `connect`、`accept`、`recv` 或 `recvfrom` 这样的阻塞程序。使用 `select` 就可以完成非阻塞方式工作的程序，它能够监视需要被监视的文件描述符的变化情况——读、写或异常。

1. select 函数原型

`select` 的函数原型是：

```
int select(int maxfdp, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct
timeval*timeout);
```

这里面用到了两个结构体：`fd_set` 和 `timeval`。结构体 `fd_set` 可以理解为一个集合，这个集合中存放的是文件描述符（file descriptor），即文件句柄，这可以认为是常说的普通意义的文件；当然 UNIX 下任何设备、管道、FIFO 等都是文件形式，所以毫无疑问，一个 socket 就是一个文件，socket 句柄就是一个文件描述符。`fd_set` 集合可以通过一些宏由人为来操作，比如以下代码：

```
fd_set set;
FD_ZERO(&set); /* 将 set 清零 */
FD_SET(fd, &set); /* 将 fd 加入 set */
FD_CLR(fd, &set); /* 将 fd 从 set 中清除 */
FD_ISSET(fd, &set); /* 如果 fd 在 set 中则真 */
```

结构体 `timeval` 是一个常用的结构，用来代表时间值，有两个成员，一个是秒数，另一个是毫秒数。

接着讲下 `select` 的各个参数所表示的含义，如下所述。

(1) `maxfdp` 是一个整数值，是指集合中所有文件描述符的范围，即所有文件描述符的最大值加 1。

(2) `readfds` 是指向 `fd_set` 结构的指针，这个集合中应该包括文件描述符。因为要监视文件描述符的读变化的，即关心是否可以从这些文件中读取数据，如果这个集合中有一个文件可读，`select` 就会返回一个大于 0 的值，表示有文件可读。如果没有可读的文件，则根据

timeout 参数再判断是否超时：若超出 timeout 的时间，select 返回 0；若发生错误返回负值；也可以传入 NULL 值，表示不关心任何文件的读变化。

(3) writefds 是指向 fd_set 结构的指针，这个集合中应该包括文件描述符。因为要监视文件描述符的写变化的，即关心是否可以向这些文件中写入数据，如果这个集合中有一个文件可写，select 就会返回一个大于 0 的值，表示有文件可写。如果没有可写的文件，则根据 timeout 参数再判断是否超时：若超出 timeout 的时间，select 返回 0；若发生错误返回负值；也可以传入 NULL 值，表示不关心任何文件的写变化。

(4) errorfds 同上面两个参数的意图，用来监视文件错误异常。

(5) timeout 是 select 的超时时间，这个参数至关重要，它可以使 select 处于 3 种状态：①若将 NULL 以形参传入，即不传入时间结构，就是将 select 置于阻塞状态，一定等到监视文件描述符集合中某个文件描述符发生变化为止；②若将时间值设为 0，就变成一个纯粹的非阻塞函数，不管文件描述符是否有变化，都立刻返回继续执行，文件无变化返回 0，有变化返回一个正值；③ timeout 的值大于 0，这就是等待的超时时间，即 select 在 timeout 时间内阻塞，超时时间之内有事件到来就返回了，否则在超时后不管怎样一定返回，返回值同上述。

(6) 返回值：准备就绪的描述符数，若超时则返回 0，若出错则返回 -1。

2. 使用 select 函数循环读取键盘输入

【例 7.1】使用 select 函数循环读取键盘输入。

keyboard.cpp 的代码是：

```
#include <sys/time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/select.h>

int main(){
    int keyboard;
    int ret,i;
    char c;
    fd_set readfd;
    struct timeval timeout;
    keyboard = open("/dev/tty",O_RDONLY | O_NONBLOCK);
    assert(keyboard>0);
```

```
while(1){
    timeout.tv_sec=1;
    timeout.tv_usec=0;
    FD_ZERO(&readfd);
    FD_SET(keyboard,&readfd);
    ret=select (keyboard+1,&readfd,NULL,NULL,&timeout);
    if(FD_ISSET(keyboard,&readfd)) {
        i=read(keyboard,&c,1);
        if ('\n'==c)
            continue;
        printf("The input is %c\n",c);
        if ('q'==c)
            break;
    }
}
return 0;
```

用 `g++ -g -o keyboard keyboard.cpp` 命令编译得到 `keyboard` 文件，执行 `./keyboard` 命令后，只要发现在键盘上键入字符，程序就输出对应的字符。执行结果如图 7-8 所示。

下面来具体看下程序里都写了哪些功能。

```
open("/dev/tty", O_RDONLY | O_NONBLOCK);
```

/dev/tty 当前终端，任何 tty（任何类型的终端设备），echo "hello" > /dev/tty 都会直接显示在当前的终端中。

O_RDONLY 指只读方式, O_NONBLOCK 指非阻塞方式。
综合起来就是, 非阻塞地读取终端上的输入信息。

```
assert (keyboard>0);
```

assert 宏的原型定义在 <assert.h> 中，其作用是如果它的条件返回错误，则终止程序执行，原型定义如下所示：

```
#include <assert.h>
void assert( int expression );
```

assert 的作用是计算表达式 expression，如果其值为假（即为 0），那么它先向 stderr 打印一条出错信息，然后通过调用 abort 来终止程序运行。

或者说，打开终端后会返回一个文件描述符，如果这个文件描述符小于等于 0，就表示打开失败，失败就得退出程序。

```
timeout.tv_sec=1;
timeout.tv_usec=0;
FD_ZERO(&readfd);
FD_SET(keyboard,&readfd);
ret=select(keyboard+1,&readfd,NULL,NULL,&timeout);
```

```

isharexu@linux 07011$ ./keyboard
a
The input is a
v
The input is v
c
The input is c
e
The input is e
l
The input is l
o
The input is o
+
The input is +
{{{
The input is {
The input is {
The input is {

```

图 7-8 例 7.1 程序执行结果图

超时时间设为 1s，把可读的 fd 集合都清空，再把打开终端的描述符加入到可读描述符集合中，调用 select 函数查看是否有可读的 fd。

```
if(FD_ISSET(keyboard,&readfd)) {
    i=read(keyboard,&c,1);
    if ('\n'==c)
        continue;
    printf("The input is %c\n",c);
    if ('q'==c)
        break;
}
```

如果终端的描述符在可读描述符集合中，就开始读取数据，如果读到的字符是 \n，即换行符，则继续；如果读到的字符是 q，就退出；否则就输出该字符。

这样就实现了循环读取了键盘的输入。但是，这里设置的超时时间似乎没有派上用场，这是由于没有判断 select 的返回值导致的。例 7.2 即把 select 的返回值做了判断，可以比较来看。

【例 7.2】 观察 select 超时时的表现。

keyboard.cpp 的代码是：

```
#include <sys/time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/select.h>

int main(){
    int keyboard;
    int ret,i;
    char c;
    fd_set readfd;
    struct timeval timeout;
    keyboard = open("/dev/tty",O_RDONLY | O_NONBLOCK);
    assert(keyboard>0);
    while(1) {
        timeout.tv_sec=5;
        timeout.tv_usec=0;
        FD_ZERO(&readfd);
        FD_SET(keyboard,&readfd);
```

```

ret=select(keyboard+1,&readfd,NULL,NULL,&timeout);
if (ret == -1)
    perror("select error");
else if (ret){
    if(FD_ISSET(keyboard,&readfd)){
        i=read(keyboard,&c,1);
        if ('\n'==c)
            continue;
        printf("hehethe input is %c\n",c);
        if ('q'==c)
            break;
    }
} else if (ret == 0)
    printf("time out\n");
}
return 0;
}

```

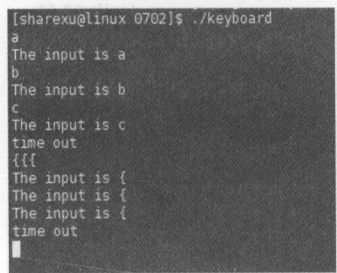
用 `g++ -g -o keyboard keyboard.cpp` 命令编译得到 `keyboard` 文件，执行 `./keyboard` 命令后，只要发现在键盘上键入字符，程序就输出对应字符。不过，如果超过 5s 不输入字符，程序会自动打印出 `time out`。执行结果如图 7-9 所示。

例 7.2 与例 7.1 相比，下面的代码有所不同。

```

if (ret == -1)
    perror("select error");
else if (ret){
    if(FD_ISSET(keyboard,&readfd)){
        i=read(keyboard,&c,1);
        if ('\n'==c)
            continue;
        printf("The input is %c\n",c);
        if ('q'==c)
            break;
    }
}
else if (ret == 0)
    printf("time out\n");
}

```



```

[sharexu@linux 0702]$ ./keyboard
a
The input is a
b
The input is b
c
The input is c
time out
{{{
The input is {
The input is {
The input is {
time out

```

图 7-9 例 7.2 程序执行结果图

`ret` 值是 `select` 函数的返回值。如果 `ret` 的值为 `-1`，即代表 `select` 函数调用失败；如果 `ret` 的值为 `0`，则代表等待时间超时，此时仍然没有可读或可写的描述符，则程序打印出 `time out`，提示已超时。

3. 使用 `select` 函数提高服务器的处理能力

【例 7.3】 使用 `select` 函数提高服务器的处理能力。

`server.cpp` 的代码如下：

```

#include <sys/types.h>
#include <sys/socket.h>

```



```

#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/wait.h>
#include <string.h>
#include <errno.h>
#define DEFAULT_PORT 6666
int main( int argc, char ** argv){
    int serverfd,acceptfd; /* 监听 socket: serverfd, 数据传输 socket: acceptfd */
    struct sockaddr_in my_addr; /* 本机地址信息 */
    struct sockaddr_in their_addr; /* 客户地址信息 */
    unsigned int sin_size, myport=6666, lisnum=10;
    if ((serverfd = socket(AF_INET , SOCK_STREAM, 0)) == -1) {
        perror("socket" );
        return -1;
    }
    printf("socket ok \n");
    my_addr.sin_family=AF_INET;
    my_addr.sin_port=htons(DEFAULT_PORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 0);
    if (bind(serverfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr ))
== -1) {
        perror("bind" );
        return -2;
    }
    printf("bind ok \n");
    if (listen(serverfd, lisnum) == -1){
        perror("listen" );
        return -3;
    }
    printf("listen ok \n");

    fd_set client_fdset; /* 监控文件描述符集合 */
    int maxsock; /* 监控文件描述符中最大的文件号 */
    struct timeval tv; /* 超时返回时间 */
    int client_sockfd[5]; /* 存放活动的 sockfd */
    bzero((void*)client_sockfd,sizeof(client_sockfd));
    int conn_amount = 0; /* 用来记录描述符数量 */
    maxsock = serverfd;
    char buffer[1024];
    int ret=0;
    while(1){
        /* 初始化文件描述符到集合 */
        FD_ZERO(&client_fdset);
        /* 加入服务器描述符 */
        FD_SET(serverfd,&client_fdset);

```

```

/* 设置超时时间 */
tv.tv_sec = 30; /*30 秒 */
tv.tv_usec = 0;
/* 把活动的句柄加入到文件描述符中 */
for(int i = 0; i < 5; ++i){
/* 程序中 Listen 中参数设为 5, 故 i 必须小于 5 */
    if(client_sockfd[i] != 0){
        FD_SET(client_sockfd[i], &client_fdset);
    }
}
/*printf("put sockfd in fdset!\n");*/
/*select 函数 */
ret = select(maxsock+1, &client_fdset, NULL, NULL, &tv);
if(ret < 0){
    perror("select error!\n");
    break;
}
else if(ret == 0){
    printf("timeout!\n");
    continue;
}
/* 轮询各个文件描述符 */
for(int i = 0; i < conn_amount; ++i){
/*FD_ISSET 检查 client_sockfd 是否可读写, >0 可读写 */
    if(FD_ISSET(client_sockfd[i], &client_fdset)){
        printf("start rcv from client[%d]:\n", i);
        ret = recv(client_sockfd[i], buffer, 1024, 0);
        if(ret <= 0){
            printf("client[%d] close\n", i);
            close(client_sockfd[i]);
            FD_CLR(client_sockfd[i], &client_fdset);
            client_sockfd[i] = 0;
        }
        else{
            printf("rcv from client[%d] :%s\n", i, buffer);
        }
    }
}
/* 检查是否有新的连接, 如果收, 接收连接, 加入到 client_sockfd 中 */
if(FD_ISSET(serverfd, &client_fdset)){
    /* 接受连接 */
    struct sockaddr_in client_addr;
    size_t size = sizeof(struct sockaddr_in);
    int sock_client = accept(serverfd, (struct sockaddr*)&client_addr, (unsigned
int*)&size);
    if(sock_client < 0){
        perror("accept error!\n");
        continue;
    }
}
/* 把连接加入到文件描述符集合中 */

```

```

if(conn_amount < 5){
    client_sockfd[conn_amount++] = sock_client;
    bzero(buffer,1024);
    strcpy(buffer, "this is server! welcome!\n");
    send(sock_client, buffer, 1024, 0);
    printf("new connection client[%d] %s:%d\n", conn_amount, inet_ntoa(client_
addr.sin_addr), ntohs(client_addr.sin_port));
    bzero(buffer,sizeof(buffer));
    ret = recv(sock_client, buffer, 1024, 0);
    if(ret < 0){
        perror("recv error!\n");
        close(serverfd);
        return -1;
    }
    printf("recv : %s\n",buffer);
    if(sock_client > maxsock){
        maxsock = sock_client;
    }
    else{
        printf("max connections!!!quit!!\n");
        break;
    }
}
}

for(int i = 0; i < 5; ++i){
    if(client_sockfd[i] != 0){
        close(client_sockfd[i]);
    }
}
close(serverfd);
return 0;
}

```

client.cpp 的代码是:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define DEFAULT_PORT 6666
int main( int argc, char * argv[]){
    int connfd = 0;
    int cLen = 0;
    struct sockaddr_in client;
    if(argc < 2){

```

```

        printf(" Usage: clientent [server IP address]\n");
        return -1;
    }
    client.sin_family = AF_INET;
    client.sin_port = htons(DEFAULT_PORT);
    client.sin_addr.s_addr = inet_addr(argv[1]);
    connfd = socket(AF_INET, SOCK_STREAM, 0);
    if(connfd < 0){
        perror("socket" );
        return -1;
    }
    if(connect(connfd, (struct sockaddr*)&client, sizeof(client)) < 0){
        perror("connect" );
        return -1;
    }
    char buffer[1024];
    bzero(buffer, sizeof(buffer));
    recv(connfd, buffer, 1024, 0);
    printf("recv : %s\n", buffer);
    bzero(buffer, sizeof(buffer));
    strcpy(buffer, "this is client!\n");
    send(connfd, buffer, 1024, 0);
    while(1){
        bzero(buffer, sizeof(buffer));
        scanf("%s", buffer);
        int p = strlen(buffer);
        buffer[p] = '\0';
        send(connfd, buffer, 1024, 0);
        printf("i have send buffer\n");
    }
    close(connfd);
    return 0;
}

```

makefile 的代码是:

```

all:server client
server:server.o
    g++ -g -o server server.o
client:client.o
    g++ -g -o client client.o
server.o:server.cpp
    g++ -g -c server.cpp
client.o:client.cpp
    g++ -g -c client.cpp
clean:all
    rm all

```

执行 make 命令后, 生成 server 和 client2 个可执行文件。分别打开 3 个终端窗口, 第一个执行 ./server 命令, 第二个执行 ./client 127.0.0.1 命令, 表示连上本机的 6666 端口。此

时第一个终端窗口里输出“new connection client[1] 127.0.0.1:36779”，若此时向第二个终端窗口中随意输入字符串，则在第一个终端窗口可以收到对应的字符串。然后在第三个终端窗口里也执行 ./client 127.0.0.1 命令，这时，第一个终端窗口会提示“new connection client[2] 127.0.0.1:36780”，同样的向第三个终端窗口输入字符串，第一个终端窗口中也可以看见。

执行结果如图 7-10 所示。

<pre>[sharexu@linux 0703]\$./server socket ok bind ok listen ok new connection client[1] 127.0.0.1:36779 recv : this is client! start recv from client[0]: recv from client[0]:hishi start recv from client[0]: recv from client[0]:haha new connection client[2] 127.0.0.1:36780 recv : this is client! start recv from client[1]: recv from client[1]:aaa start recv from client[1]: recv from client[1]:bbb start recv from client[1]: recv from client[1]:ccc start recv from client[0]: recv from client[0]:hh start recv from client[0]: recv from client[0]:hehe ^C [sharexu@linux 0703]\$</pre>	<pre>:hello server client recv msg is:hello server client recv msg is:hello server client recv msg is:hello server client recv msg is:hello server client recv msg is:hello server client recv msg is:hello server ^C [sharexu@linux 0701]\$ a -bash: a: command not found [sharexu@linux 0701]\$ cd .. [sharexu@linux chapter07]\$ cd 0703 [sharexu@linux 0703]\$./client.127.0.0.1 recv : this is server! welcome! hishi i have send buffer haha i have send buffer hh"H i have send buffer hehe i have send buffer ^C [sharexu@linux 0703]\$</pre>	<pre>[sharexu@linux ~]\$ cd chapter07/0701/ [sharexu@linux 0701]\$./client 127.0.0.1 client send to server client recv msg is:hello server client recv msg is:hello server client recv msg is:hello server client recv msg is:hello server client recv msg is:hello server client recv msg is:hello server client recv msg is:hello server ^C [sharexu@linux 0701]\$ cd ../ [sharexu@linux chapter07]\$ cd 0703 [sharexu@linux 0703]\$./client 127.0.0.1 recv : this is server! welcome! aaa i have send buffer bbb i have send buffer ccc i have send buffer h i have send buffer j [sharexu@linux 0703]\$</pre>
--	---	--

图 7-10 例 7.3 程序执行结果图

下面来具体分析程序各部分内容。

server.cpp 中，获得监听描述符后，就开始了一个 while 循环。在这个 while 循环中，需要不断地查看是否有新 client 连接；已连上的 client 是否有发送消息过来。先初始化文件描述符集合，把服务器描述符加入到集合中，设置 select 的超时时间，代码如下所示。

```
/* 初始化文件描述符到集合 */
FD_ZERO(&client_fdset);
/* 加入服务器描述符 */
FD_SET(serverfd,&client_fdset);
/* 设置超时时间 */
tv.tv_sec = 30; /*30 秒 */
tv.tv_usec = 0;
```

把已连上的 client 的 fd 也加入到集合中，方便检查是否有数据可读，代码如下：

```
/* 把活动的句柄加入到文件描述符中 */
for(int i = 0; i < 5; ++i){/* 程序中 Listen 中参数设为 5，故 i 必须小于 5*/
    if(client_sockfd[i] != 0){
        FD_SET(client_sockfd[i], &client_fdset);
    }
}
```

调用 select 函数，注意要根据起返回值判断程序是否有异常，代码如下：

```

ret = select(maxsock+1, &client_fdset, NULL, NULL, &tv);
if(ret < 0){
    perror("select error!\n");
    break;
}
else if(ret == 0){
    printf("timeout!\n");
    continue;
}

```

先看已连上的 client 的 fd 有无可读的数据，也就是有无数据可接收的，有就输出，没有或异常时，要将相应的 client 关闭连接，并将它在集合里清掉，以关闭这个 fd，代码如下：

```

for(int i = 0; i < conn_amount; ++i){
    /*FD_ISSET 检查 client_sockfd 是否可读，>0 可读写 */
    if(FD_ISSET(client_sockfd[i], &client_fdset)){
        printf("start recv from client[%d]:\n", i);
        ret = recv(client_sockfd[i], buffer, 1024, 0);
        if(ret <= 0){
            printf("client[%d] close\n", i);
            close(client_sockfd[i]);
            FD_CLR(client_sockfd[i], &client_fdset);
            client_sockfd[i] = 0;
        }
        else{
            printf("recv from client[%d] :%s\n", i, buffer);
        }
    }
}

```

检查是否有新的连接，如果有，建立接收连接，加入到 client_sockfd 中，发送一个消息给 client，方便看到 client 已经连上了。并且，还需要把 maxsock 更新，因为下一次进入 while 循环调用 select 时，需要传当前最大的 fd 值 +1 给 select 函数。相关代码如下所示。

```

if(FD_ISSET(serverfd, &client_fdset)){
    /* 接受连接 */
    struct sockaddr_in client_addr;
    size_t size = sizeof(struct sockaddr_in);
    int sock_client = accept(serverfd, (struct sockaddr*)&client_addr, (unsigned
int*)&size);
    if(sock_client < 0){
        perror("accept error!\n");
        continue;
    }
    /* 把连接加入到文件描述符集合中 */
    if(conn_amount < 5){
        client_sockfd[conn_amount++] = sock_client;
        bzero(buffer, 1024);
        strcpy(buffer, "this is server! welcome!\n");
    }
}

```



```

    send(sock_client, buffer, 1024, 0);
    printf("new connection client[%d] %s:%d\n", conn_amount, inet_ntoa(client_
addr.sin_addr), ntohs(client_addr.sin_port));
    bzero(buffer, sizeof(buffer));
    ret = recv(sock_client, buffer, 1024, 0);
    if(ret < 0){
        perror("recv error!\n");
        close(serverfd);
        return -1;
    }
    printf("recv : %s\n", buffer);
    if(sock_client > maxsock){
        maxsock = sock_client;
    }
    else{
        printf("max connections!!!quit!!\n");
        break;
    }
}
}
}

```

最后，把已连上的 client 的 fd 和 server 自身的 fd 都关闭，代码如下：

```

for(int i = 0; i < 5; ++i){
    if(client_sockfd[i] != 0){
        close(client_sockfd[i]);
    }
}
close(serverfd);

```

如此，server 就能同时处理多个 client 的请求，达到提供处理能力的效果。

而 client.cpp 中，也是连上 server 后开始发数据，但这部分比较简单，读者可自行研究。

7.3 poll

1. poll 函数原型

和 select 函数一样，poll 函数也可以用于执行多路复用 IO。

poll 所需要的头文件和函数原型如下所示：

```

#include<poll.h>
int poll(struct pollfd * fds, unsigned int nfds, int timeout);

```

pollfd 结构体定义如下所示：

```

struct pollfd {
    int fd; /* 文件描述符 */
    short events; /* 等待的事件 */

```

```
short revents; /* 实际发生了的事件 */
};
```

每一个 pollfd 结构体指定了一个被监视的文件描述符，可以传递多个结构体，指示 poll() 监视多个文件描述符。每个结构体的 events 域是监视该文件描述符的事件掩码，由用户来设置这个域的属性。revents 域是文件描述符的操作结果事件掩码，内核在调用返回时设置这个域；并且 events 域中请求的任何事件都可能在 revents 域中返回。具体的事件代码和代表的含义如表 7-1 所示。

表 7-1 poll 事件

事件分类	事件代码	意义
合法事件	POLLIN	有数据可读
	POLLRDNORM	有普通数据可读
	POLLRDBAND	有优先数据可读
	POLLPRI	有紧迫数据可读
	POLLOUT	写数据不会导致阻塞
	POLLWRNORM	写普通数据不会导致阻塞
	POLLWRBAND	写优先数据不会导致阻塞
	POLLMSGSIGPOLL	消息可用
非法事件	POLLER	指定的文件描述符发生错误
	POLLHUP	指定的文件描述符挂起事件
	POLLNVAL	指定的文件描述符非法

实际上这些事件在 events 域中无意义，因为它们总会在合适的时候从 revents 中返回。

使用 poll() 和 select() 不一样，不需要显式地请求异常情况报告。

POLLIN | POLLPRI 等价于 select() 的读事件，POLLOUT | POLLWRBAND 等价于 select() 的写事件；POLLIN 等价于 POLLRDNORM | POLLRDBAND，而 POLLOUT 则等价于 POLLWRNORM。例如，要同时监视一个文件描述符是否可读或可写，可以设置 events 为 POLLIN | POLLOUT。在 poll 返回时，只要检查 revents 中的标志，获得对应于文件描述符请求的 events 结构体。如果 POLLIN 事件被设置，则文件描述符可以被读取而不阻塞；如果 POLLOUT 被设置，则文件描述符可以写入而不导致阻塞。这些标志并不是互斥的：它们可能被同时设置，表示这个文件描述符的读取和写入操作都会正常返回而不阻塞。

timeout 参数指定等待的毫秒数，无论 IO 是否准备好，poll 都会返回。timeout 指定为负数值时表示无限超时，使 poll() 一直挂起直到一个指定事件发生；timeout 为 0 指示 poll 调用立即返回并列出准备好 IO 的文件描述符，但并不等待其他的事件。这种情况下，poll() 的返回值，一旦被选举出来，立即返回。

成功时，poll() 返回结构体中 revents 域不为 0 的文件描述符个数；如果在超时前没有任何事件发生，poll() 返回 0。失败时，poll() 返回 -1，并设置 errno 为下列值之一。

- (1) EBADF: 一个或多个结构体中指定的文件描述符无效。
- (2) EFAULTfds: 指针指向的地址超出进程的地址空间。
- (3) EINTR: 请求的事件之前产生一个信号, 调用可以重新发起。
- (4) EINVALfds: 参数超出 PLIMIT_NOFILE 值。
- (5) ENOMEM: 可用内存不足, 无法完成请求。

2. 使用 poll 函数提高服务器处理能力

【例 7.4】使用 poll 函数提高服务器处理能力。

server.cpp 的代码是:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/wait.h>
#include <string.h>
#include <errno.h>
#include <poll.h>

#define IPADDRESS "127.0.0.1"
#define PORT 6666
#define MAXLINE 1024
#define LISTENQ 5
#define OPEN_MAX 1000
#define INFTIM -1

/* 创建套接字, 进行绑定和监听 */
int bind_and_listen(){
    int serverfd; /* 监听 socket: serverfd */
    struct sockaddr_in my_addr; /* 本机地址信息 */
    unsigned int sin_size;
    if ((serverfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return -1;
    }
    printf("socket ok \n");
    my_addr.sin_family=AF_INET;
    my_addr.sin_port=htons(PORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 0);
    if (bind(serverfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) ==
-1) {
        perror("bind");
        return -2;
    }
}
```

```

printf("bind ok \n");
if (listen(serverfd, LISTENQ) == -1) {
    perror("listen");
    return -3;
}
printf("listen ok \n");
return serverfd;
}

/*IO 多路复用 poll*/
void do_poll(int listenfd){
    int connfd, sockfd;
    struct sockaddr_in cliaddr;
    socklen_t cliaddrlen;
    struct pollfd clientfds[OPEN_MAX];
    int maxi;
    int i;
    int nready;
    /* 添加监听描述符 */
    clientfds[0].fd = listenfd;
    clientfds[0].events = POLLIN;
    /* 初始化客户连接描述符 */
    for (i = 1; i < OPEN_MAX; i++)
        clientfds[i].fd = -1;
    maxi = 0;
    /* 循环处理 */
    while(1){
        /* 获取可用描述符的个数 */
        nready = poll(clientfds, maxi+1, INFTIM);
        if (nready == -1){
            perror("poll error:");
            exit(1);
        }
        /* 测试监听描述符是否准备好 */
        if (clientfds[0].revents & POLLIN){
            cliaddrlen = sizeof(cliaddr);
            /* 接受新的连接 */
            if ((connfd = accept(listenfd, (struct sockaddr*)&cliaddr, &cliaddrlen))
                == -1){
                if (errno == EINTR)
                    continue;
                else{
                    perror("accept error:");
                    exit(1);
                }
            }
            fprintf(stdout, "accept a new client: %s:%d\n", inet_ntoa(cliaddr.sin_addr),
                cliaddr.sin_port);
            /* 将新的连接描述符添加到数组中 */
            for (i = 1; i < OPEN_MAX; i++){

```

```

    if (clientfds[i].fd < 0){
        clientfds[i].fd = connfd;
        break;
    }
}
if (i == OPEN_MAX){
    fprintf(stderr, "too many clients.\n");
    exit(1);
}
/* 将新的描述符添加到读描述符集合中 */
clientfds[i].events = POLLIN;
/* 记录客户连接套接字的个数 */
maxi = (i > maxi ? i : maxi);
if (--nready <= 0)
    continue;
}
/* 处理多个连接上客户端发来的包 */
char buf[MAXLINE];
memset(buf, 0, MAXLINE);
int readlen=0;
for (i = 1; i <= maxi; i++){
    if (clientfds[i].fd < 0)
        continue;
    /* 测试客户描述符是否准备好 */
    if (clientfds[i].revents & POLLIN){
        /* 接收客户端发送的信息 */
        readlen = read(clientfds[i].fd, buf, MAXLINE);
        if (readlen == 0){
            close(clientfds[i].fd);
            clientfds[i].fd = -1;
            continue;
        }
        /* printf("read msg is: "); */
        write(STDOUT_FILENO, buf, readlen);
        /* 向客户端发送 buf */
        write(clientfds[i].fd, buf, readlen);
    }
}
}

int main(int argc, char *argv[]){
    int listenfd=bind_and_listen();
    if(listenfd<0){
        return 0;
    }
    do_poll(listenfd);
    return 0;
}

```

client.cpp 的代码是:

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <sys/wait.h>
#include <string.h>
#include <errno.h>
#include <poll.h>
#define MAXLINE 1024
#define DEFAULT_PORT 6666
#define max(a,b) (a > b) ? a : b
static void handle_connection(int sockfd);
int main(int argc, char *argv[]){
    int connfd = 0;
    int cLen = 0;
    struct sockaddr_in client;
    if(argc < 2){
        printf("Usage: clientent [server IP address]\n");
        return -1;
    }
    client.sin_family = AF_INET;
    client.sin_port = htons(DEFAULT_PORT);
    client.sin_addr.s_addr = inet_addr(argv[1]);
    connfd = socket(AF_INET, SOCK_STREAM, 0);
    if(connfd < 0){
        perror("socket");
        return -1;
    }
    if(connect(connfd, (struct sockaddr*)&client, sizeof(client)) < 0){
        perror("connect");
        return -1;
    }
    /* 处理连接描述符 */
    handle_connection(connfd);
    return 0;
}

static void handle_connection(int sockfd){
    char    sendline[MAXLINE], recvline[MAXLINE];
    int     maxfdp, stdineof;
    struct pollfd pfd[2];
    int n;
    /* 添加连接描述符 */
    pfd[0].fd = sockfd;
    pfd[0].events = POLLIN;
    /* 添加标准输入描述符 */
    pfd[1].fd = STDIN_FILENO;

```



```

pfds[1].events = POLLIN;
while(1){
    poll(pfds,2,-1);
    if (pfds[0].revents & POLLIN){
        n = read(sockfd,recvline,MAXLINE);
        if (n == 0){
            fprintf(stderr,"client: server is closed.\n");
            close(sockfd);
        }
        write(STDOUT_FILENO,recvline,n);
    }
    /* 测试标准输入是否准备好 */
    if (pfds[1].revents & POLLIN) {
        n = read(STDIN_FILENO,sendline,MAXLINE);
        if (n == 0) {
            shutdown(sockfd,SHUT_WR);
            continue;
        }
        write(sockfd,sendline,n);
    }
}
}

```

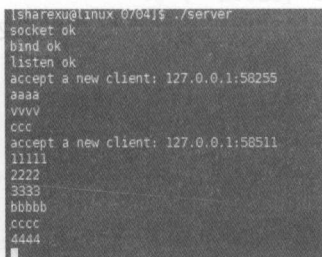
makefile 的代码是:

```

all:server client
server:server.o
    g++ -g -o server server.o
client:client.o
    g++ -g -o client client.o
server.o:server.cpp
    g++ -g -c server.cpp
client.o:client.cpp
    g++ -g -c client.cpp
clean:all
    rm all

```

程序执行结果如图 7-11、图 7-12 和图 7-13 所示。

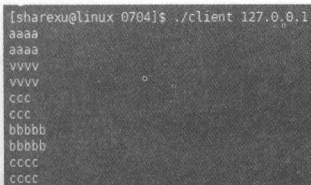


```

[sharexu@linux 0704]$ ./server
socket ok
bind ok
listen ok
accept a new client: 127.0.0.1:58255
aaaa
vvvv
ccc
accept a new client: 127.0.0.1:58511
11111
2222
3333
bbbbbb
cccc
4444

```

图 7-11 例 7.4 服务器端执行结果图

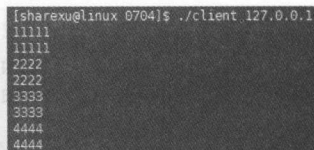


```

[sharexu@linux 0704]$ ./client 127.0.0.1
aaaa
vvvv
ccc
11111
2222
3333
bbbbbb
cccc
4444

```

图 7-12 例 7.4 客户端 1 执行结果图



```

[sharexu@linux 0704]$ ./client 127.0.0.1
11111
2222
3333
3333
4444
4444

```

图 7-13 例 7.4 客户端 2 执行结果图

例 7.4 中，编写了一个 echo server 程序，功能是客户端向服务器发送消息，服务器接收输出并原样返回客户端，客户端接收到消息后输出到终端。

server.cpp 中，把套接字的创建、绑定和监听，都写在了一个函数中，方便阅读。

```
int bind_and_listen(){
    int serverfd; /* 监听 socket: serverfd */
    struct sockaddr_in my_addr; /* 本机地址信息 */
    unsigned int sin_size;
    if ((serverfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return -1;
    }
    printf("socket ok \n");
    my_addr.sin_family=AF_INET;
    my_addr.sin_port=htons(PORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 0);
    if (bind(serverfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) ==
-1) {
        perror("bind");
        return -2;
    }
    printf("bind ok \n");
    if (listen(serverfd, LISTENQ) == -1) {
        perror("listen");
        return -3;
    }
    printf("listen ok \n");
    return serverfd;
}
```

先把服务器的描述符加入到描述符集合中，需要注意的是，select 所用的描述符集合是一个 fd_set 的结构体中，而 poll 的描述符却是在一个以 pollfd 为元素的数组中代码如下：

```
/* 添加监听描述符 */
clientfds[0].fd = listenfd;
clientfds[0].events = POLLIN;
```

接下来将数组初始化，注意别把第一个元素给覆盖了，因为第一个已添加了服务器描述符。所以，i 是从 1 开始，而不是从 0 开始，代码如下：

```
/* 初始化客户连接描述符 */
for (i = 1; i < OPEN_MAX; i++)
    clientfds[i].fd = -1;
```

接着是一个 while 循环，查看是否有新客户端连接，或者老客户端是否有数据发送过来。这里的超时时间设为 -1，表示无限超时，使 poll() 一直挂起直到一个指定事件发生。而 maxi 就是 clientfds 数组中最大的下标。有新 client 接入时，判断是否放在了比较大的下标位置，

是的话要修改 maxi 的值，如果放在了已经断掉连接的下标位置，则不用更新，代码如下：

```
nready = poll(clientfds,maxi+1,INFTIM);
if (nready == -1){
    perror("poll error:");
    exit(1);
}
```

当有新的客户端连接时，必须接受，获得新的 fd，并将新 fd 放到数组中，代码如下：

```
if (clientfds[0].revents & POLLIN){
    cliaddrlen = sizeof(cliaddr);
    /* 接受新的连接 */
    if ((connfd = accept(listenfd,(struct sockaddr*)&cliaddr,&cliaddrlen)) == -1){
        if (errno == EINTR)
            continue;
        else{
            perror("accept error:");
            exit(1);
        }
    }
    fprintf(stdout,"accept a new client: %s:%d\n", inet_ntoa(cliaddr.sin_addr),cliaddr.
sin_port);
    /* 将新的连接描述符添加到数组中 */
    for (i = 1;i < OPEN_MAX;i++){
        if (clientfds[i].fd < 0){
            clientfds[i].fd = connfd;
            break;
        }
    }
    if (i == OPEN_MAX){
        fprintf(stderr,"too many clients.\n");
        exit(1);
    }
    /* 将新的描述符添加到读描述符集合中 */
    clientfds[i].events = POLLIN;
    /* 记录客户连接套接字的个数 */
    maxi = (i > maxi ? i : maxi);
    if (--nready <= 0)
        continue;
}
```

还需要处理已连接上来的客户端有可能发来的包，代码如下：

```
char buf[MAXLINE];
memset(buf,0,MAXLINE);
int readlen=0;
for (i = 1;i <= maxi;i++){
    if (clientfds[i].fd < 0)
        continue;
    /* 测试客户描述符是否准备好 */
```

```

if (clientfds[i].revents & POLLIN){
    /* 接收客户端发送的信息 */
    readlen = read(clientfds[i].fd,buf,MAXLINE);
    if (readlen == 0){
        close(clientfds[i].fd);
        clientfds[i].fd = -1;
        continue;
    }
    /*printf("read msg is: ");*/
    write(STDOUT_FILENO,buf,readlen);
    /* 向客户端发送 buf */
    write(clientfds[i].fd,buf,readlen);
}
}

```

例 7.4 的 client.cpp 值得仔细看一下，笔者也用了 poll 函数进行读写操作。判断是否有数据可读，需要检查两个来源：①服务器是否发来了包；②标准输入中是否有输入。代码如下所示。

```

struct pollfd pfd[2];
int n;
/* 添加连接描述符 */
pfd[0].fd = sockfd;
pfd[0].events = POLLIN;
/* 添加标准输入描述符 */
pfd[1].fd = STDIN_FILENO;
pfd[1].events = POLLIN;
while (1){
    poll(pfd,2,-1);
    if (pfd[0].revents & POLLIN) {
        n = read(sockfd,recvline,MAXLINE);
        if (n == 0) {
            fprintf(stderr,"client: server is closed.\n");
            close(sockfd);
        }
        write(STDOUT_FILENO,recvline,n);
    }
    /* 测试标准输入是否准备好 */
    if (pfd[1].revents & POLLIN){
        n = read(STDIN_FILENO,sendline,MAXLINE);
        if (n == 0) {
            shutdown(sockfd,SHUT_WR);
            continue;
        }
        write(sockfd,sendline,n);
    }
}
}

```

综上所述，poll 函数也可让服务器具备同时处理多个客户端请求的能力。

7.4 epoll

epoll是在Linux 2.6内核中提出的，是之前select和poll的增强版本。相对于select和poll来说，epoll更加灵活，没有描述符限制。epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间之间的数据拷贝只需一次。

1. epoll 接口

使用epoll必须包含下面的这个头文件：

```
#include <sys/epoll.h>
```

epoll操作过程需要3个接口，分别如下：

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int
timeout);
```

下面分别介绍这3个接口的功能、入参和出参的含义。

```
int epoll_create(int size);
```

创建一个epoll的句柄，size用来告诉内核要监听的数目。这个参数不同于select()中的第一个参数，是最大监听的fd+1的值。需要注意的是，当创建好epoll句柄后，它就会占用一个fd值，在Linux下如果查看/proc/进程的id/fd/，是能够看到这个fd值的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

epoll的事件注册函数，它不同于select()在监听事件时告诉内核要监听什么类型的事件，而是先注册要监听的事件类型。第一个参数是epoll_create()的返回值，第二个参数表示动作，用3个宏来表示：①EPOLL_CTL_ADD，注册新的fd到epfd中；②EPOLL_CTL_MOD，修改已经注册的fd的监听事件；③EPOLL_CTL_DEL：从epfd中删除一个fd。

第3个参数是需要监听的fd，第4个参数是告诉内核需要监听什么事，struct epoll_event结构如下：

```
struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

events可以是以下几个宏的集合：①EPOLLIN，表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；②EPOLLOUT，表示对应的文件描述符可以写；③EPOLLPRI，表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；④EPOLLERR，

表示对应的文件描述符发生错误；⑤ EPOLLHUP，表示对应的文件描述符被挂断；⑥ EPOLLET，将 EPOLL 设为边缘触发（Edge Triggered）模式，这是相对于水平触发（Level Triggered）来说的；⑦ EPOLLONESHOT，只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个 socket 的话，需要再次把这个 socket 加入到 EPOLL 队列里。

```
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int
timeout);
```

等待事件的产生，类似于 select() 调用。参数 events 用来从内核得到事件的集合，maxevents 告诉内核这个 events 有多大，且 maxevents 的值不能大于创建 epoll_create() 时的 size，参数 timeout 是超时时间（ms 为单位，0 会立即返回，-1 将不确定或称永久阻塞）。该函数返回需要处理的事件数目，如返回 0 表示已超时。

2. 用 epoll 提高服务器的处理能力

下面 epoll 编写一个服务器，处理多个 client 的请求。

【例 7.5】 用 epoll 处理服务器和客户端的读写。

server.cpp 的代码是：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/epoll.h>
#include <unistd.h>
#include <sys/types.h>

#define IPADDRESS "127.0.0.1"
#define PORT 6666
#define MAXSIZE 1024
#define LISTENQ 5
#define FDSIZE 1000
#define EPOLLEVENTS 100

/* 函数声明 */
/* 创建套接字并进行绑定 */
int socket_bind(const char* ip,int port);
/*IO 多路复用 epoll*/
void do_epoll(int listenfd);
/* 事件处理函数 */
void handle_events(int epollfd,struct epoll_event *events,int num,int
listenfd,char *buf);
/* 处理接收到的连接 */
void handle_accpet(int epollfd,int listenfd);
```



```

/* 读处理 */
void do_read(int epollfd, int fd, char *buf);
/* 写处理 */
void do_write(int epollfd, int fd, char *buf);
/* 添加事件 */
void add_event(int epollfd, int fd, int state);
/* 修改事件 */
void modify_event(int epollfd, int fd, int state);
/* 删除事件 */
void delete_event(int epollfd, int fd, int state);

int main(int argc, char *argv[]) {
    int listenfd;
    listenfd = socket_bind(IPADDRESS, PORT);
    listen(listenfd, LISTENQ);
    do_epoll(listenfd);
    return 0;
}

int socket_bind(const char* ip, int port) {
    int listenfd;
    struct sockaddr_in servaddr;
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1) {
        perror("socket error:");
        exit(1);
    }
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, ip, &servaddr.sin_addr);
    servaddr.sin_port = htons(port);
    if (bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1) {
        perror("bind error: ");
        exit(1);
    }
    return listenfd;
}

void do_epoll(int listenfd) {
    int epollfd;
    struct epoll_event events[EPOLLEVENTS];
    int ret;
    char buf[MAXSIZE];
    memset(buf, 0, MAXSIZE);
    /* 创建一个描述符 */
    epollfd = epoll_create(FDSIZE);
    /* 添加监听描述符事件 */
    add_event(epollfd, listenfd, EPOLLIN);
    while(1) {
        /* 获取已经准备好的描述符事件 */

```

```

    ret = epoll_wait(epollfd, events, EPOLLEVENTS, -1);
    handle_events(epollfd, events, ret, listenfd, buf);
}
close(epollfd);
}

void handle_events(int epollfd, struct epoll_event *events, int num, int
listenfd, char *buf){
    int i;
    int fd;
    /* 进行选好遍历 */
    for (i = 0; i < num; i++){
        fd = events[i].data.fd;
        /* 根据描述符的类型和事件类型进行处理 */
        if ((fd == listenfd) && (events[i].events & EPOLLIN))
            handle_accpet(epollfd, listenfd);
        else if (events[i].events & EPOLLIN)
            do_read(epollfd, fd, buf);
        else if (events[i].events & EPOLLOUT)
            do_write(epollfd, fd, buf);
    }
}

void handle_accpet(int epollfd, int listenfd){
    int clifd;
    struct sockaddr_in cliaddr;
    socklen_t cliaddrlen;
    clifd = accept(listenfd, (struct sockaddr*)&cliaddr, &cliaddrlen);
    if (clifd == -1)
        perror("accpet error:");
    else{
        printf("accept a new client: %s:%d\n", inet_ntoa(cliaddr.sin_addr), cliaddr.
sin_port);
        /* 添加一个客户描述符和事件 */
        add_event(epollfd, clifd, EPOLLIN);
    }
}

void do_read(int epollfd, int fd, char *buf){
    int nread;
    nread = read(fd, buf, MAXSIZE);
    if (nread == -1){
        perror("read error:");
        close(fd);
        delete_event(epollfd, fd, EPOLLIN);
    }
    else if (nread == 0){
        fprintf(stderr, "client close.\n");
        close(fd);
        delete_event(epollfd, fd, EPOLLIN);
    }
}

```

```

    }
    else{
        printf("read message is : %s",buf);
        /* 修改描述符对应的事件, 由读改为写 */
        modify_event(epollfd,fd,EPOLLOUT);
    }
}

```

```

void do_write(int epollfd,int fd,char *buf){
    int nwrite;
    nwrite = write(fd,buf,strlen(buf));
    if (nwrite == -1){
        perror("write error:");
        close(fd);
        delete_event(epollfd,fd,EPOLLOUT);
    }
    else
        modify_event(epollfd,fd,EPOLLIN);
    memset(buf,0,MAXSIZE);
}

```

```

void add_event(int epollfd,int fd,int state){
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd,EPOLL_CTL_ADD,fd,&ev);
}

```

```

void delete_event(int epollfd,int fd,int state){
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd,EPOLL_CTL_DEL,fd,&ev);
}

```

```

void modify_event(int epollfd,int fd,int state){
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd,EPOLL_CTL_MOD,fd,&ev);
}

```

客户端也用 epoll 实现, 控制 STDIN_FILENO、STDOUT_FILENO 和 sockfd 3 个描述符。
client.cpp 的代码是:

```

#include <netinet/in.h>
#include <sys/socket.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

```

```

#include <sys/epoll.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>
#include <arpa/inet.h>

#define MAXSIZE 1024
#define IPADDRESS "127.0.0.1"
#define SERV_PORT 6666
#define FDSIZE 1024
#define EPOLLEVENTS 20

void handle_connection(int sockfd);
void handle_events(int epollfd, struct epoll_event *events, int num, int
sockfd, char *buf);
void do_read(int epollfd, int fd, int sockfd, char *buf);
void do_read(int epollfd, int fd, int sockfd, char *buf);
void do_write(int epollfd, int fd, int sockfd, char *buf);
void add_event(int epollfd, int fd, int state);
void delete_event(int epollfd, int fd, int state);
void modify_event(int epollfd, int fd, int state);
int count=0;
int main(int argc, char *argv[]){
    int sockfd;
    struct sockaddr_in servaddr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    inet_pton(AF_INET, IPADDRESS, &servaddr.sin_addr);
    connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
    /* 处理连接 */
    handle_connection(sockfd);
    close(sockfd);
    return 0;
}

void handle_connection(int sockfd){
    int epollfd;
    struct epoll_event events[EPOLLEVENTS];
    char buf[MAXSIZE];
    int ret;
    epollfd = epoll_create(FDSIZE);
    add_event(epollfd, STDIN_FILENO, EPOLLIN);
    while (1) {
        ret = epoll_wait(epollfd, events, EPOLLEVENTS, -1);
        handle_events(epollfd, events, ret, sockfd, buf);
    }
    close(epollfd);
}

```

```

void handle_events(int epollfd, struct epoll_event *events, int num, int
sockfd, char *buf){
    int fd;
    int i;
    for (i = 0; i < num; i++){
        fd = events[i].data.fd;
        if (events[i].events & EPOLLIN)
            do_read(epollfd, fd, sockfd, buf);
        else if (events[i].events & EPOLLOUT)
            do_write(epollfd, fd, sockfd, buf);
    }
}

```

```

void do_read(int epollfd, int fd, int sockfd, char *buf){
    int nread;
    nread = read(fd, buf, MAXSIZE);
    if (nread == -1){
        perror("read error:");
        close(fd);
    }
    else if (nread == 0){
        fprintf(stderr, "server close.\n");
        close(fd);
    }
    else{
        if (fd == STDIN_FILENO)
            add_event(epollfd, sockfd, EPOLLOUT);
        else{
            delete_event(epollfd, sockfd, EPOLLIN);
            add_event(epollfd, STDOUT_FILENO, EPOLLOUT);
        }
    }
}

```

```

void do_write(int epollfd, int fd, int sockfd, char *buf){
    int nwrite;
    char temp[100];
    buf[strlen(buf)-1] = '\0';
    snprintf(temp, sizeof(temp), "%s_%02d\n", buf, count++);
    nwrite = write(fd, temp, strlen(temp));
    if (nwrite == -1){
        perror("write error:");
        close(fd);
    }
    else{
        if (fd == STDOUT_FILENO)
            delete_event(epollfd, fd, EPOLLOUT);
        else
            modify_event(epollfd, fd, EPOLLIN);
    }
    memset(buf, 0, MAXSIZE);
}

```

```

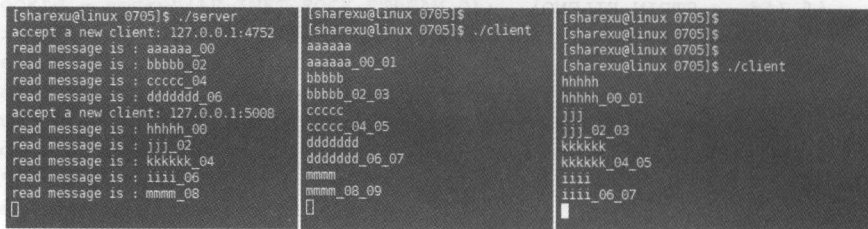
void add_event(int epollfd,int fd,int state){
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd,EPOLL_CTL_ADD,fd,&ev);
}

void delete_event(int epollfd,int fd,int state){
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd,EPOLL_CTL_DEL,fd,&ev);
}

void modify_event(int epollfd,int fd,int state){
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd,EPOLL_CTL_MOD,fd,&ev);
}

```

程序的执行结果如图 7-14 所示：



```

[sharexu@linux 0705]$ ./server
accept a new client: 127.0.0.1:4752
read message is : aaaaaa_00
read message is : bbbbb_02
read message is : ccccc_04
read message is : ddddddd_06
accept a new client: 127.0.0.1:5008
read message is : hhhhh_00
read message is : jjj_02
read message is : kkkkkk_04
read message is : llll_06
read message is : mmmm_08
[]

[sharexu@linux 0705]$ ./client
aaaaaa
aaaaaa_00_01
bbbbbb
bbbbbb_02_03
cccccc
cccccc_04_05
ddddddd
ddddddd_06_07
mmmmmm
mmmmmm_08_09
[]

[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$ ./client
hhhhh
hhhhh_00_01
jjj
jjj_02_03
kkkkkk
kkkkkk_04_05
llll
llll_06_07
[]

```

图 7-14 例 7.5 程序执行结果图

程序将各个模块的功能写在各个函数中，这样代码阅读起来会更清晰。

server.cpp 中，主函数 main 中只有 3 个动作：绑定套接字、监听套接字以及利用 epoll 处理读写操作。

```

int main(int argc,char *argv[]){
    int listenfd;
    listenfd = socket_bind(IPADDRESS,PORT);
    listen(listenfd,LISTENQ);
    do_epoll(listenfd);
    return 0;
}

```

绑定套接字与监听套接字这个知识点在前面例子中多有使用，这里不再赘述。下面重

点介绍如何使用 `epoll` 函数。首先需要声明一个以结构体 `epoll_event` 为元素的数组，对比 `select` 和 `poll` 可以看到，`select` 用的是 `fd_set` 的结构体，`poll` 用的是以结构体 `pollfd` 为元素的数组，3 个函数使用的结构都不一样。先把监听描述符加入到事件列表中，`select` 和 `poll` 也是先把监听描述符加入到各自的结构中。这里还要再创建 `epollfd`，这个设置与 `select` 和 `poll` 都不一样。

```
void do_epoll(int listenfd){
    int epollfd;
    struct epoll_event events[EPOLLEVENTS];
    int ret;
    char buf[MAXSIZE];
    memset(buf,0,MAXSIZE);
    /* 创建一个描述符 */
    epollfd = epoll_create(FDSIZE);
    /* 添加监听描述符事件 */
    add_event(epollfd,listenfd,EPOLLIN);
    while(1){
        /* 获取已经准备好的描述符事件 */
        ret = epoll_wait(epollfd,events,EPOLLEVENTS,-1);
        handle_events(epollfd,events,ret,listenfd,buf);
    }
    close(epollfd);
}
```

获取到已经准备好的描述符事件后，根据描述符的类型和事件类型进行处理。如果描述符是监听描述符且事件可读时，就可以接收新的客户端请求；如果事件仅为可读，则读取老客户发过来的包；如果事件为可写，则发包。

```
void handle_events(int epollfd,struct epoll_event *events,int num,int
listenfd,char *buf){
    int i;
    int fd;
    /* 进行选好遍历 */
    for (i = 0;i < num;i++){
        fd = events[i].data.fd;
        /* 根据描述符的类型和事件类型进行处理 */
        if ((fd == listenfd) && (events[i].events & EPOLLIN))
            handle_accpet(epollfd,listenfd);
        else if (events[i].events & EPOLLIN)
            do_read(epollfd,fd,buf);
        else if (events[i].events & EPOLLOUT)
            do_write(epollfd,fd,buf);
    }
}
```

由于这里设计的是服务器会根据客户端发来的内容同样地回包，所以读到数据后，要把事件转为可写状态，由写事件进行发包。读操作失败了或结束后，就需要关闭相应的 `fd`，并把读状态的事件删除。

```

void do_read(int epollfd,int fd,char *buf){
    int nread;
    nread = read(fd,buf,MAXSIZE);
    if (nread == -1){
        perror("read error:");
        close(fd);
        delete_event(epollfd,fd,Epollin);
    }
    else if (nread == 0){
        fprintf(stderr,"client close.\n");
        close(fd);
        delete_event(epollfd,fd,Epollin);
    }
    else{
        printf("read message is : %s",buf);
        /* 修改描述符对应的事件，由读改为写 */
        modify_event(epollfd,fd,Epollout);
    }
}

```

若写操作失败了，要将其从写事件列表里删除；若写操作成功了，还要继续检测是否有数据，有的话继续读入。

```

void do_write(int epollfd,int fd,char *buf){
    int nwrite;
    nwrite = write(fd,buf,strlen(buf));
    if (nwrite == -1){
        perror("write error:");
        close(fd);
        delete_event(epollfd,fd,Epollout);
    }
    else
        modify_event(epollfd,fd,Epollin);
    memset(buf,0,MAXSIZE);
}

```

添加、删除、修改事件，都是用 `epoll_ctl` 函数，只是第二个参数不同，分别是 `EPOLL_CTL_ADD`、`EPOLL_CTL_DEL`、`EPOLL_CTL_MOD`，各自代码如下：

```

epoll_ctl(epollfd,EPOLL_CTL_ADD,fd,&ev);
epoll_ctl(epollfd,EPOLL_CTL_DEL,fd,&ev);
epoll_ctl(epollfd,EPOLL_CTL_MOD,fd,&ev);

```

`client.cpp` 中，则需要关注三种状态：①标准输入；②服务器发来了数据；③标准输出（把服务器的东西发过来的打印到标准输出上）。客户端中第一个加入 `epollfd` 的是 `STDIN_FILENO`，即标准输入状态，因为是等待客户端输入字符串，服务器才会回应相应的字符串。

处理读事件要注意，由于读事件有两个触发点：标准输入和服务器发来了数据。所以，如果是从标准输入读来的数据，要接着去发给服务器，要加一个写状态的事件，并且还要继

续往标准输入里输入东西，所以这个读事件不改变状态。如果是从服务器读来的数据，则要将其输出到标准输出中，并把读事件删除，加上写事件。

```
void do_read(int epollfd,int fd,int sockfd,char *buf){
    int nread;
    nread = read(fd,buf,MAXSIZE);
    if (nread == -1){
        perror("read error:");
        close(fd);
    }
    else if (nread == 0){
        fprintf(stderr,"server close.\n");
        close(fd);
    }
    else{
        if (fd == STDIN_FILENO)
            add_event(epollfd,sockfd,EPOLOUT);
        else{
            delete_event(epollfd,sockfd,EPOLLIN);
            add_event(epollfd,STDOUT_FILENO,EPOLOUT);
        }
    }
}
```

处理写事件时也要注意，有两种写事件：向标准输出里写或向网络中写（给服务器发包）。如果是往标准输出里写，操作结束就应删除事件；但是如果向网络中写，操作结束后仍要继续读服务器有可能发过来的数据，并要改变事件状态。

```
void do_write(int epollfd,int fd,int sockfd,char *buf){
    int nwrite;
    char temp[100];
    buf[strlen(buf)-1]='\0';
    snprintf(temp,sizeof(temp),"%s_%02d\n",buf,count++);
    nwrite = write(fd,temp,strlen(temp));
    if (nwrite == -1){
        perror("write error:");
        close(fd);
    }
    else{
        if (fd == STDOUT_FILENO)
            delete_event(epollfd,fd,EPOLOUT);
        else
            modify_event(epollfd,fd,EPOLLIN);
    }
    memset(buf,0,MAXSIZE);
}
```

从执行结果中可以看出，比如客户端给服务器发了“aaaaaa_00”，客户端最终输出到标准输出的是“aaaaaa_00_01”，这也证明了给服务器发包和输出数据到标准输出时，用的是

write 函数 (count 只有在调用 write 函数时才会自增), 如图 7-15 所示。

```

[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$ ./server
accept a new client: 127.0.0.1:4752
read message 15 : aaaaa_00
read message 15 : hbbbb_02
read message 15 : ccccc_04
read message 15 : dddddd_06
accept a new client: 127.0.0.1:5008
read message 15 : hhhhh_00
read message 15 : jjj_02
read message 15 : kkkkk_04
read message 15 : llll_06
read message 15 : mmmm_08
client close.
client close.
^C
[sharexu@linux 0705]$

[sharexu@linux 0705]$
bbbb_02_03
^C
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$
[sharexu@linux 0705]$ ./client
aaaaa
aaaaa_00_01
hhbbb
bbbb_02_03
cccc
cccc_04_05
dddddd
dddddd_06_07
mmmm
mmmm_08_09
^C
[sharexu@linux 0705]$

```

图 7-15 例 7.5 程序执行结果图

3. select、poll 和 epoll 的区别

select、poll 和 epoll 都是多路 IO 复用的机制。多路 IO 复用就通过一种机制, 可以监视多个描述符, 一旦某个描述符就绪 (一般是读就绪或者写就绪), 能够通知程序进行相应的读写操作。但 select、poll 和 epoll 本质上都是同步 IO, 因为它们都需要在读写事件就绪后自己负责进行读写, 即是阻塞的, 而异步 IO 则无须自己负责进行读写, 异步 I/O 的实现会负责把数据从内核拷贝到用户空间。

下面对这 3 种多路 IO 复用进行对比。

(1) 首先还是来看常见的 select() 和 poll()。对于网络编程来说, 一般认为 poll() 比 select() 要高级一些, 这主要源于以下几个原因。

1) poll() 不要求开发者在计算最大文件描述符时进行 +1 的操作。

2) poll() 在应付大数目的文件描述符的时候速度更快, 因为对于 select() 来说内核需要检查大量描述符对应的 fd_set 中的每一个比特位, 比较费时。

3) select() 可以监控的文件描述符数目是固定的, 相对来说也较少 (1024 或 2048)。如果需要监控数值比较大的文件描述符, 或是分布得很稀疏的较少的描述符, 效率也会很低。而对于 poll() 函数来说, 就可以创建特定大小的数组来保存监控的描述符, 而不受文件描述符值大小的影响, 而且 poll() 可以监控的文件数目远大于 select()。

4) 对于 select() 来说, 所监控的 fd_set 在 select() 返回之后会发生变化, 所以在下一次进入 select() 之前都需要重新初始化需要监控的 fd_set, poll() 函数将监控的输入和输出事件分开, 允许被监控的文件数组被复用而不需要重新初始化。

5) select() 函数的超时参数在返回时也是未定义的, 考虑到可移植性, 每次在超时之后在下次进入到 select() 之前都需要重新设置超时参数。

(2) `select()` 的优点如下所述。

1) `select()` 的可移植性更好,在某些 UNIX 系统上不支持 `poll()`。

2) `select()` 对于超时值提供了更好的精度,而 `poll()` 是精度较差。

(3) `epoll` 的优点如下所述。

1) 支持一个进程打开大数目的 `socket` 描述符 (FD)。

`select()` 最不能忍受的是一个进程所打开的 FD 是有一定限制的,由 `FD_SETSIZE` 的默认值是 1024/2048。对于那些需要支持上万连接数目的 IM 服务器来说显然太少了。这时候可以选择修改这个宏然后重新编译内核。不过 `epoll` 则没有这个限制,它所支持的 FD 上限是最大可以打开文件的数目,这个数字一般远大于 2048。举个例子,在 1GB 内存的空间中这个数字一般是 10 万左右,具体数目可以使用 `cat/proc/sys/fs/file-max` 查看,一般来说这个数目和系统内存关系很大。

2) IO 效率不随 FD 数目增加而线性下降。

传统的 `select/poll` 另一个致命弱点就是当你拥有一个很大的 `socket` 集合,不过由于网络延迟,任一时间只有部分的 `socket` 是“活跃”的,但是 `select/poll` 每次调用都会线性扫描全部的集合,导致效率呈现线性下降。但是 `epoll` 不存在这个问题,它只会对“活跃”的 `socket` 进行操作——这是因为在内核中实现 `epoll` 是根据每个 fd 上面的 `callback` 函数实现的。那么,只有“活跃”的 `socket` 才会主动去调用 `callback` 函数,其他 `idle` 状态 `socket` 则不会,在这点上, `epoll` 实现了一个“伪”AIO,因为这时候推动力由 Linux 内核提供。

3) 使用 `mmap` 加速内核与用户空间的消息传递。

这点实际上涉及 `epoll` 的具体实现。无论是 `select`、`poll` 还是 `epoll` 都需要内核把 fd 消息通知给用户空间,如何避免不必要的内存拷贝就显得尤为重要。在这点上, `epoll` 是通过内核与用户空间 `mmap` 处于同一块内存实现的。

对于 `poll` 来说需要将用户传入的 `pollfd` 数组拷贝到内核空间,因为拷贝操作和数组长度相关,时间上来看,这是一个 $O(n)$ 操作,当事件发生后, `poll` 将获得的数据传送到用户空间,并执行释放内存和剥离等待队列等工作,向用户空间拷贝数据与剥离等待队列等操作的时间复杂度同样是 $O(n)$ 。

7.5 本章小结

本章讲述了网络 IO 模型, `select`、`poll` 和 `epoll` 用法,可以用这些模型或函数提供服务器的并发处理能力,以节约机器成本。

第 8 章会继续学习网络的常用分析工具。

网络分析工具

前面介绍了程序网络间的通信，但在处理定位问题或者多方联调时，网络分析工具往往会派上用场。本章主要讲 ping、tcpdump、netstat 和 lsof 这 4 个网络分析工具的使用。

8.1 ping

1. ping 介绍

ping (Packet Internet Groper, 因特网包探索器) 是 Windows、UNIX 和 Linux 系统下的一个命令。ping 也属于一个通信协议，是 TCP/IP 协议的一部分。利用 ping 命令可以检查网络是否连通，可以很好地帮助分析和判定网络故障。应用格式：ping 空格 IP 地址，该命令还可以加许多参数使用，如图 8-1 所示。

```
[sharexu@linux ~]$ ping
Usage: ping [-L RbdfnrvVa] [-c count] [-i interval] [-w deadline]
          [-p pattern] [-s packetsize] [-t ttl] [-I interface or address]
          [-M mtu discovery hint] [-S sndbuf]
          [-T timestamp option] [-Q tos] [hop1 ...] destination
[sharexu@linux ~]$
```

图 8-1 执行 ping 命令可以看到可使用的参数

ping 发送一个 ICMP (Internet Control Messages Protocol, 因特网信报控制协议)，请求消息给目的地并报告是否收到所希望的 ICMP echo (ICMP 回声应答)，它是用来检查网络是否通畅或者网络连接速度的命令。作为一个后台开发者，ping 是一个必须掌握的命令。它所利用的原理是这样的：利用网络上机器 IP 地址的唯一性，给目标 IP 地址发送一个数据

包，再要求对方返回一个同样大小的数据包来确定两台网络机器是否连接相通以及时延是多少。

ping 指的是端对端连通，通常用来作为可用性的检查，但是某些病毒木马会强行大量远程执行 ping 命令来抢占你的网络资源，导致系统、网速变慢。严禁 ping 入侵作为大多数防火墙的一个基本功能提供给用户进行选择。通常的情况下你如果不用作服务器或者进行网络测试，可以放心的选中它，以保护你的计算机。

2. ping 的使用

网络连通问题是由许多原因引起的，如本地配置错误、远程主机协议失效等，当然还包括设备等造成的故障。使用 ping 检查连通性有以下 6 个步骤。

- (1) 使用 ipconfig/all 观察本地网络设置是否正确。
- (2) ping 127.0.0.1，来检查本地的 TCP/IP 协议有没有设置好，如图 8-2 所示。
- (3) ping 本机 IP 地址，这样是为了检查本机的 IP 地址是否设置有误。
- (4) ping 本网网关或本网 IP 地址，这样的是为了检查硬件设备是否有问题，也可以检查本机与本地网络连接是否正常。（在非局域网中这一步骤可以忽略）
- (5) ping 本地 DNS 地址，这样做是为了检查本地 DNS 服务器是否工作正常。
- (6) ping 远程 IP 地址，这主要是检查本网或本机与外部的连接是否正常。ping 远程 IP 地址还可以用来测试网络延时。比如输入“ping www.baidu.com”（百度域名）之后屏幕会显示如图 8-3 所示的信息。

```
[sharexu@linux ~]$ ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data:
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.030 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.026 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.031 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.021 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.030 ms
64 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=0.021 ms
^C
--- 127.0.0.1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 538ms
rtt min/avg/max/mdev = 0.021/0.026/0.031/0.006 ms
[sharexu@linux ~]$
```

图 8-2 ping 127.0.0.1 检查本地的 TCP/IP 协议有没有设置好

```
[sharexu@linux ~]$ ping www.baidu.com
PING www.a.shifen.com (115.239.210.27) 56(84) bytes of data:
64 bytes from 115.239.210.27: icmp_seq=1 ttl=52 time=15.7 ms
64 bytes from 115.239.210.27: icmp_seq=2 ttl=52 time=15.7 ms
64 bytes from 115.239.210.27: icmp_seq=3 ttl=52 time=15.7 ms
64 bytes from 115.239.210.27: icmp_seq=4 ttl=52 time=15.7 ms
64 bytes from 115.239.210.27: icmp_seq=5 ttl=52 time=15.7 ms
64 bytes from 115.239.210.27: icmp_seq=6 ttl=52 time=15.7 ms
64 bytes from 115.239.210.27: icmp_seq=7 ttl=52 time=15.7 ms
^C
--- www.a.shifen.com ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 611ms
rtt min/avg/max/mdev = 15.712/15.744/15.796/0.137 ms
[sharexu@linux ~]$
```

图 8-3 ping 百度域名会发出的包

图 8-3 中可以看到，向百度域名发了 7 个包，对方 7 个都收到了，没有丢包，共耗时 6113ms，RTT（一个连接的往返时间）的最小、平均、最大和算术平均差分别是 15.712ms、15.744ms、15.796ms 和 0.137。

后面的 time=15.7ms 是响应时间，这个时间越小，说明连接的这个地址速度越快。

从图 8-4 也可以看到，用 ping 命令还可以寻找固定网站网址的 IP 地址。其中的 115.239.210.27 也是 www.baidu.com 其中的一台机器的 IP 地址，在浏览器输入这个地址，发现也跳转到了百度首页。

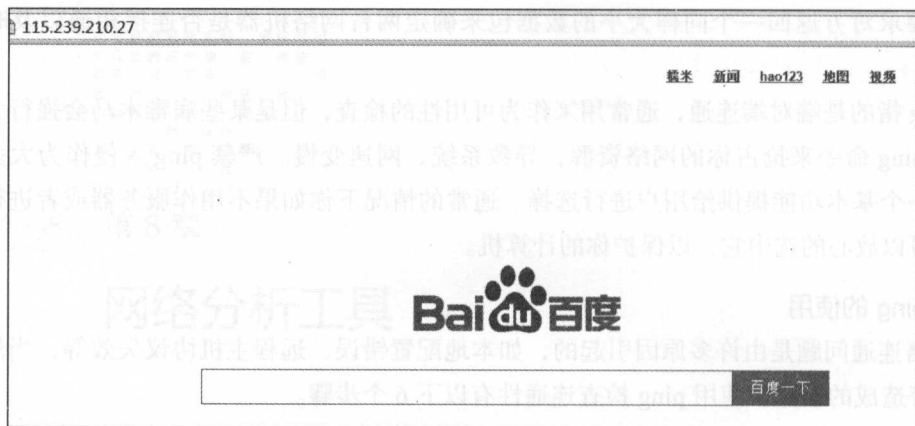


图 8-4 在浏览器中访问 ping 得到的 IP 地址

Linux 的 ping 语法和 Windows 的差不多，但是 Linux 的 ping 数据包是 64Byte，而 Windows 的是 32Byte，Windows 下默认发送 4 次数据包后结束，Linux 下的 ping 程序默认不停发送数据包，直到用户手动停止（停止指令是 Ctrl+c）。

8.2 tcpdump

1. tcpdump 介绍

tcpdump 可以将网络中传送的数据包的“头”完全截获下来提供分析。它支持针对协议、主机、网络或端口的过滤，并提供 and、or、not 等逻辑语句来帮助去掉无用的信息。简单来说，tcpdump 就是一种免费的网络分析工具，而且它提供了源代码，公开了接口，因此具备很强的可扩展性，对于网络维护和防止入侵都是非常有用的工具。tcpdump 存在于基本的 FreeBSD 系统中，由于它需要将网络界面设置为混杂模式，普通用户不能正常执行，但具备 root 权限的用户可以直接执行它来获取网络上的信息。因此系统中存在网络分析工具主要不会对本机安全产生威胁，而是会对网络上的其他计算机的安全产生威胁。

tcpdump 根据使用者的定义对网络上的数据包进行截获和分析。作为互联网上经典的系统管理工具，tcpdump 以其强大的功能，灵活的截取策略，成为每个高级的系统管理员分析网络、排查问题等所必备的工具之一。

tcpdump 支持相当多的不同参数，如使用 -i 参数指定 tcpdump 监听的网络界面，这在计算机具有多个网络界面时非常有用；使用 -c 参数指定要监听的数据包数量，使用 -w 参数指定将监听到的数据包写入文件中保存，等等。

然而更复杂的 tcpdump 参数是用于过滤操作，这是因为网络中流量很大，如果不加

分辨地将所有的数据包都截留下来,可能会因为数据量太大,反而不容易发现需要的数据包。使用这些参数定义的过滤规则可以截留特定的数据包,以缩小目标,才能更好地分析网络中存在的问题。tcpdump 使用参数指定要监视数据包的类型、地址、端口等,根据具体的网络问题,充分利用这些过滤规则就能达到迅速定位故障的目的。下面将介绍如何使用 tcpdump。

2. tcpdump 使用

tcpdump 采用命令行方式,它的命令格式为:

```
tcpdump [ -adeflnNOPqStvx ] [ -c 数量 ] [ -F 文件名 ]
        [ -i 网络接口 ] [ -r 文件名 ] [ -s snaplen ]
        [ -T 类型 ] [ -w 文件名 ] [ 表达式 ]
```

表达式是一个正则表达式, tcpdump 利用它作为过滤报文的条件,如果一个报文满足表达式的条件,则这个报文将会被捕获。如果没有给出任何条件,则网络上所有的信息包将会被截获。在表达式中一般包含如下几种类型的关键字。

1) 关于类型的关键字,主要包括 host、net、port 等,例如 host 210.27.48.2 即指明 210.27.48.2 是一台主机, net 202.0.0.0 指明 202.0.0.0 是一个网络地址, port 23 指明端口号是 23。如果没有指定类型,默认的类型是 host。

2) 确定传输方向的关键字,主要包括 src、dst、dst or src、dst、src 等,这些关键字指明了传输的方向。举例说明, src 210.27.48.2, 指明 IP 包中源地址是 210.27.48.2, dst net 202.0.0.0 指明目的网络地址是 202.0.0.0。如果没有指明方向关键字,则默认是 src or dst 关键字。

3) 协议的关键字,主要包括 fddi、ip、arp、rarp、tcp、udp 等类型。fddi 指明是在 FDDI (分布式光纤数据接口网络) 上的特定的网络协议,实际上它是 ether 的别名, fddi 和 ether 具有类似的源地址和目的地址,所以可以将 fddi 协议包当作 ether 的包进行处理和分析。其他的几个关键字就是指明了监听的包的协议内容。如果没有指定任何协议,则 tcpdump 将会监听所有协议的信息包。

除了这 3 种类型的关键字之外,其他重要的关键字如下: gateway、broadcast、less、greater 等,还有 3 种逻辑运算: 取非运算 not/!, 与运算 and/&& ; 或运算 or/| ; 这些关键字可以组合起来构成强大的组合条件来满足人们的需要,下面举几个例子来说明。

1) 截取某主机相关的包。

a. 想要截获所有 210.27.48.1 的主机收到的和发出的所有的数据包,使用如下命令:

```
tcpdump host 210.27.48.1
```

b. 想要截获主机 210.27.48.1 和主机 210.27.48.2 或 210.27.48.3 的通信,使用如下命令:
(在命令行中使用括号时,一定要添加 “\”)

```
tcpdump host 210.27.48.1 and \ (210.27.48.2 or 210.27.48.3 \)
```

c. 如果想要获取主机 210.27.48.1 除了和主机 210.27.48.2 之外所有主机通信的 ip 包，使用如下命令：

```
tcpdump ip host 210.27.48.1 and ! 210.27.48.2
```

d. 如果想要获取主机 210.27.48.1 接收或发出的 telnet 包，使用如下命令：

```
tcpdump tcp port 23 host 210.27.48.1
```

2) 截取某端口相关的包。

如果想要获取在端口 6666 上通过的包，使用如下命令：

```
tcpdump port 6666
```

3) 截取某网卡的包。

如果想要获取在网卡 eth1 上通过的包，使用如下命令：

```
tcpdump -ieth1
```

8.3 netstat

netstat 命令用于显示与 IP、TCP、UDP 和 ICMP 协议相关的统计数据，一般用于检验本机各端口的网络连接情况。netstat 是在内核中访问网络及相关信息的程序，它能提供 TCP 连接、对 TCP 和 UDP 的监听及获取进程内存管理的相关报告。

计算机有时候会因接收到的数据报导致出错数据或故障，TCP/IP 可以容许这些类型的错误，并能够自动重发数据报。但如果累计的出错情况数目占到所接收的 IP 数据报相当大的百分比，或者它的数目正迅速增加，那么就on应该使用 netstat 查一查为什么会出现这些情况了。

netstat 的命令格式如下所示：

```
netstat [-acCeFghilMnNoprstuvVwx] [-A< 网络类型 >] [--ip]
```

netstat 用于显示与 IP、TCP、UDP 和 ICMP 协议相关的统计数据，一般用于检验本机各端口的网络连接情况。

执行 netstat 命令后，输出结果如图 8-5 所示。

从整体上看，netstat 的输出结果可以分为两个部分：① Active Internet connections，称为有源 TCP 连接，其中 Recv-Q 和 Send-Q 指的是接收队列和发送队列，这些数字一般都是 0，如果不是则表示请求包和回包正在队列中堆积；② Active UNIX domain sockets，称为有源 UNIX 域套接口（和网络套接字一样，但是只能用于本机通信，性能可以提高一倍）。

```
[sharexu@linux ~]$ netstat
Active Internet connections (w/o servers)

```

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	2384	0	42.96.142.129:smtp	101.130.131.206:enpc	ESTABLISHED
tcp	0	52	42.96.142.129:ssh	116.24.141.4:51222	ESTABLISHED

```

Active UNIX domain sockets (w/o servers)

```

Proto	RefCnt	Flags	Type	State	I-Node	Path
unix	7	[]	DGRAM		7724	/dev/log
unix	2	[]	DGRAM		6707	@/org/kernel/udev/udevfd
unix	3	[]	STREAM	CONNECTED	582367	
unix	3	[]	STREAM	CONNECTED	582366	
unix	2	[]	DGRAM		582363	
unix	2	[]	STREAM	CONNECTED	208787	
unix	2	[]	STREAM	CONNECTED	208697	
unix	2	[]	STREAM	CONNECTED	208485	
unix	2	[]	STREAM	CONNECTED	207295	
unix	3	[]	STREAM	CONNECTED	8139	
unix	3	[]	STREAM	CONNECTED	8138	
unix	3	[]	STREAM	CONNECTED	8136	
unix	3	[]	STREAM	CONNECTED	8135	
unix	3	[]	STREAM	CONNECTED	8126	
unix	2	[]	STREAM	CONNECTED	8125	
unix	3	[]	STREAM	CONNECTED	8124	
unix	3	[]	STREAM	CONNECTED	8123	
unix	2	[]	DGRAM		7908	
unix	2	[]	DGRAM		7890	
unix	2	[]	DGRAM		7869	
unix	2	[]	DGRAM		7809	
unix	3	[]	DGRAM		6723	
unix	3	[]	DGRAM		6722	

```
[sharexu@linux ~]$
```

图 8-5 执行 netstat 命令查看网络端口数据

Proto 显示连接使用的协议，RefCnt 表示连接到本套接口上的进程号，Types 显示套接口的类型，State 显示套接口当前的状态，Path 表示连接到套接口的其他进程使用的路径名。

常见参数如下所示：

- a (all) 显示所有选项，默认不显示 LISTEN 相关
- t (tcp) 仅显示 tcp 相关选项
- u (udp) 仅显示 udp 相关选项
- n 拒绝显示别名，能显示数字的全部转化成数字。
- l 仅列出有在 Listen (监听) 的服务状态
- p 显示建立相关链接的程序名
- r 显示路由信息，路由表
- e 显示扩展信息，例如 uid 等
- s 按各个协议进行统计
- c 每隔一个固定时间，执行该 netstat 命令。

提示：LISTEN 和 LISTENING 的状态只有用 -a 或者 -l 才能看到

各参数的使用实例如下所述。

- (1) 列出所有端口 (包括监听和未监听的): netstat -a。
- (2) 列出所有 TCP 端口: netstat -at。
- (3) 列出所有 UDP 端口: netstat -au。
- (4) 列出所有处于监听状态的 socket: netstat -l。
- (5) 列出所有监听 TCP 端口的 socket: netstat -lt。
- (6) 列出所有监听 UDP 端口的 socket: netstat -lu。
- (7) 列出所有监听 UNIX 端口的 socket: netstat -lx。
- (8) 在 netstat 输出中显示 PID 和进程名称: netstat -p。
- (9) 当你不想让主机，端口和用户名显示，使用 netstat -n，将会使用数字代替那些名称。

(10) 持续输出 netsta 信息: `netstat -c`。

(11) 找出程序运行的端口: `netstat -ap | grep ssh`。

(12) 找出运行在指定端口的进程, 如 `netstat -an | grep ':80'`。

(13) 显式网络接口列表: `netstat -i`。

(14) IP 和 TCP 的分析, 如查看链接某服务端口最多的 IP 地址命令是:

```
netstat -nat | grep "192.168.1.15:22" | awk '{print $5}' | awk -F: '{print $1}' | sort | uniq -c | sort -nr | head -20。
```

(15) TCP 各自状态列表: `netstat -nat | awk '{print $6}'`。

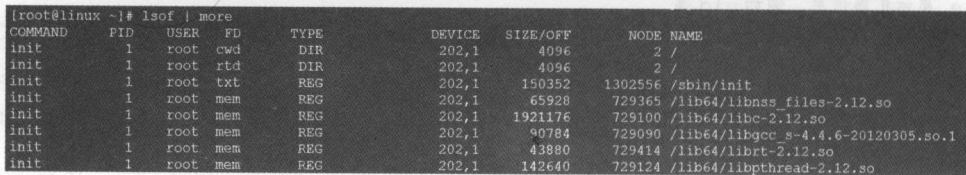
(16) 先把各种 TCP 状态全都取出来, 然后使用 `uniq -c` 统计, 之后再排序: `netstat -nat | awk '{print $6}' | sort | uniq -c`。

8.4 lsof

`lsof` (list open file) 是一个列出当前系统打开文件的工具。在 Linux 环境下, 任何事物都以文件的形式存在, 通过文件不仅仅可以访问常规数据, 还可以访问网络连接和硬件。所以如传输控制协议 (TCP) 和用户数据报协议 (UDP) 套接字等, 系统在后台都为该应用程序分配了一个文件描述符, 无论这个文件的本质如何, 该文件描述符为应用程序与基础操作系统之间的交互提供了通用接口。因为应用程序打开文件的描述符列表提供了大量关于这个应用程序本身的信息, 因此通过 `lsof` 工具能够查看这个列表对系统监测以及排错将是很有帮助的。

在终端下输入 `lsof` 即可显示系统打开的文件, 因为 `lsof` 需要访问核心内存和各种文件, 所以必须以 `root` 用户的身份运行才能够充分地发挥其功能。

执行 `lsof` 命令后, 输出结果如图 8-6 所示。



COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE NAME
init	1	root	cwd	DIR	202,1	4096	2 /
init	1	root	rtd	DIR	202,1	4096	2 /
init	1	root	txt	REG	202,1	150352	1302596 /sbin/init
init	1	root	mem	REG	202,1	65928	729365 /lib64/libnss_files-2.12.so
init	1	root	mem	REG	202,1	1921176	729100 /lib64/libc-2.12.so
init	1	root	mem	REG	202,1	90784	729090 /lib64/libgcc_s-4.4.6-20120305.so.1
init	1	root	mem	REG	202,1	43880	729414 /lib64/librt-2.12.so
init	1	root	mem	REG	202,1	142640	729124 /lib64/libpthread-2.12.so

图 8-6 执行 `lsof` 命令结果图

(1) 每行显示一个打开的文件, 若不指定条件默认将显示所有进程打开的所有文件。`lsof` 输出各列信息的意义如下所述。

1) **COMMAND**: 进程的名称。

2) **PID**: 进程标识符。

3) **USER**: 进程所有者。

4) **FD**: 文件描述符, 应用程序通过文件描述符识别该文件如 `cwd`、`txt` 等。

5) **TYPE**: 文件类型, 如 `DIR`、`REG` 等。

- 6) DEVICE: 指定磁盘的名称。
- 7) SIZE: 文件的大小。
- 8) NODE: 索引节点 (文件在磁盘上的标识)。
- 9) NAME: 打开文件的确切名称。

(2) lsof 语法格式是:

```
lsof [ options ] filename
```

(3) 常用的参数列表:

```
lsof filename 显示打开指定文件的所有进程
lsof -a 表示两个参数都必须满足时才显示结果
lsof -c string 显示 COMMAND 列中包含指定字符的进程所有打开的文件
lsof -u username 显示所属 user 进程打开的文件
lsof -g gid 显示归属 gid 的进程情况
lsof +d /DIR/ 显示目录下被进程打开的文件
lsof +D /DIR/ 同上, 但是会搜索目录下的所有目录, 时间相对较长
lsof -d FD 显示指定文件描述符的进程
lsof -n 不将 IP 转换为 hostname, 缺省是不加上 -n 参数
lsof -i 用以显示符合条件的进程情况
```

(4) 常用命令如下所述。

1) 查看 6666 端口现在运行情况, 命令: `lsof -i :6666`, 结果如图 8-7 所示。

```
[root@linux ~]# lsof -i :6666
COMMAND  PID  USER  FD   TYPE DEVICE SIZE/OFF NODE NAME
server  29862 sharexu 3u    IPv4 885165      0t0  TCP *:ircu-2 (LISTEN)
```

图 8-7 执行 `lsof -i :6666` 命令结果图

2) 查看所属 root 用户进程所打开的文件, 文件类型为 .txt: `lsof -a -u root -d txt`, 结果如图 8-8 所示。

```
[root@linux ~]# lsof -a -u root -d txt
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF  NODE NAME
init       1 root  txt    REG 202,1  150352 1302956 /sbin/init
kthreadd   2 root  txt    unknown          /proc/2/exe
migration  3 root  txt    unknown          /proc/3/exe
ksftirqd   4 root  txt    unknown          /proc/4/exe
migration  5 root  txt    unknown          /proc/5/exe
```

图 8-8 执行 `lsof -a -u root -d txt` 命令结果图

3) 监控打开的文件和设备。查看设备 `/dev/tty1` 被哪些进程占用的命令是: `lsof /dev/tty1`, 结果如图 8-9 所示。

```
[root@linux ~]# lsof /dev/tty1
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF  NODE NAME
mingetty 1192 root    0u   CHR  4,1      0t0 5063 /dev/tty1
mingetty 1192 root    1u   CHR  4,1      0t0 5063 /dev/tty1
mingetty 1192 root    2u   CHR  4,1      0t0 5063 /dev/tty1
```

图 8-9 执行 `lsof /dev/tty1` 命令结果图

4) 监控程序。如查看指定程序 server 打开的文件：lsdf -c server，结果如图 8-10 所示。

```
[root@linux ~]# lsdf -c server
COMMAND  PID  USER  FD  TYPE  DEVICE  SIZE/OFF  NODE NAME
server   29890 sharexu cwd   DIR   202,1    4096    262254 /home/sharexu/chartpter06/0605
server   29890 sharexu rtd   DIR   202,1    4096     2 /
server   29890 sharexu txt   REG   202,1    13670   262263 /home/sharexu/chartpter06/0605/server
server   29890 sharexu mem   REG   202,1   1921176 729100 /lib64/libc-2.12.so
server   29890 sharexu mem   REG   202,1    90784   729090 /lib64/libgcc_s-4.4.6-20120305.so.1
server   29890 sharexu mem   REG   202,1   596272 729347 /lib64/libm-2.12.so
server   29890 sharexu mem   REG   202,1   987096 672481 /usr/lib64/libstdc++.so.6.0.13
server   29890 sharexu mem   REG   202,1   154624 729317 /lib64/ld-2.12.so
server   29890 sharexu 0u   CHR   136,1     0t0     4 /dev/pts/1
server   29890 sharexu 1u   CHR   136,1     0t0     4 /dev/pts/1
server   29890 sharexu 2u   CHR   136,1     0t0     4 /dev/pts/1
server   29890 sharexu 3u   IPv4   587995     0t0     TCP *:ircu-2 (LISTEN)
```

图 8-10 执行 lsdf -c server 命令结果图

5) 监控用户。比如查看指定用户 sharexu 打开的文件：lsdf -u sharexu，结果如图 8-11 所示。

```
[root@linux ~]# lsdf -u sharexu
COMMAND  PID  USER  FD  TYPE  DEVICE  SIZE/OFF  NODE NAME
sshd     29775 sharexu cwd   DIR   202,1    4096     2 /
sshd     29775 sharexu rtd   DIR   202,1    4096     2 /
sshd     29775 sharexu txt   REG   202,1   530104 673437 /usr/sbin/sshd
sshd     29775 sharexu DEL   REG     0,4      582356 /dev/zero
sshd     29775 sharexu mem   REG   202,1   18592 729338 /lib64/security/pam_limits.so
sshd     29775 sharexu mem   REG   202,1   10224 729336 /lib64/security/pam_keyinit.so
```

图 8-11 执行 lsdf -u sharexu 命令结果图

8.5 本章小结

本章主要介绍了 ping、tcpdump、netstat 和 lsdf 这 4 个工具的常用用法，可以帮助分析网络情况。

接下来的第 9 章主要讲另外一个提高服务器处理能力的方法：多线程。对多线程概念的掌握和理解水平，也常被认为是衡量一个人的编程实力的重要参考指标。

多线程

为了更好地理解多线程的概念，先对进程、线程的概念背景做一下简单介绍。早期的计算机系统都只允许一个程序独占系统资源，一次只能执行一个程序。在大型机年代，计算能力是一种宝贵资源，对于资源拥有方来说，最好的生财之道自然是将同一资源同时租售给尽可能多的用户。最理想的情况是垄断全球计算市场，以至于当年 IBM 都预测“全球只要有 4 台计算机就够了”。

这种背景下，一台计算机能够支持多个程序并发执行的需求变得十分迫切，由此产生了进程的概念。进程在多数早期多任务操作系统中是执行工作的基本单元。进程是包含程序指令和相关资源的集合，每个进程和其他进程一起参与调度，竞争 CPU、内存等系统资源。每次进程切换，都存在进程资源的保存和恢复动作，这称为上下文切换。进程的引入可以解决多用户支持的问题，但是多进程系统也在如下方面产生了新的问题：进程频繁切换引起的额外开销可能会严重影响系统性能。进程间通信要求复杂的系统级实现。在程序功能日趋复杂的情况下，上述缺陷也就凸显出来。

比如，一个简单的 GUI 程序，为了有更好的交互性，通常用一个任务支持界面交互，另一个任务支持后台运算。如果每个任务均由一个进程来实现，那会相当低效，但对每个进程来说，系统资源看上去都是其独占的，比如内存空间，每个进程都认为自己的内存空间是独有的。每一次切换，这些独立资源都需要切换。由此就演化出了利用分配给同一个进程的资源，尽量实现多个任务的方法，这也就引入了线程的概念。

同一个进程内部的多个线程，共享的是同一个进程的所有资源。比如，与每个进程拥有自己的内存空间不同，同属一个进程的多个线程共享该进程的内存空间。例如在进程地址空间中有一个全局变量 `GlobalVar`，若 A 线程将其赋值为 1，则另一线程 B 可以看

到该变量值为 1，但两个线程看到的全局变量 GlobalVar 本质是同一个变量。通过线程可以支持同一个应用程序内部的并发，免去了进程频繁切换的开销，另外并发任务间通信也更简单。

网络程序具有天生的并发性，比如网络数据库可能需要同时处理数以千计的请求。而由于网络连接的时延具有不确定性和不可靠性，一旦等待一次网络交互时，可以让当前线程进入睡眠，退出调度，而去处理其他线程。这样就能够有效利用系统资源，充分发挥系统实时处理能力。线程的切换是轻量级的，所以可以保证足够快。每当有事件发生状态改变，都能有线程及时响应，而且每次线程内部处理的计算强度和复杂度都不大。在这种情况下，多线程实现的模型也是高效的。

9.1 多线程是什么

一个程序的运行过程中，只有一个控制权的存在。当函数被调用的时候，该函数获得控制权，成为激活（active）函数，然后运行该函数中的指令。与此同时，其他的函数处于离场状态，并不运行，如图 9-1 所示。

图 9-1 中，各个方块之间由箭头连接，而各个函数就像是连在一根线上一样，计算机像一条流水线一样执行各个函数中定义的操作。这样的程序叫做单线程程序。

多线程就是允许一个进程内存在多个控制权，以便让多个函数同时处于激活状态，从而让多个函数的操作同时运行。即使是单核 CPU 的计算机，也可以通过不停地不同线程的指令间切换，从而造成多线程同时运行的效果。图 9-2 所示为一个多线程的流程图。

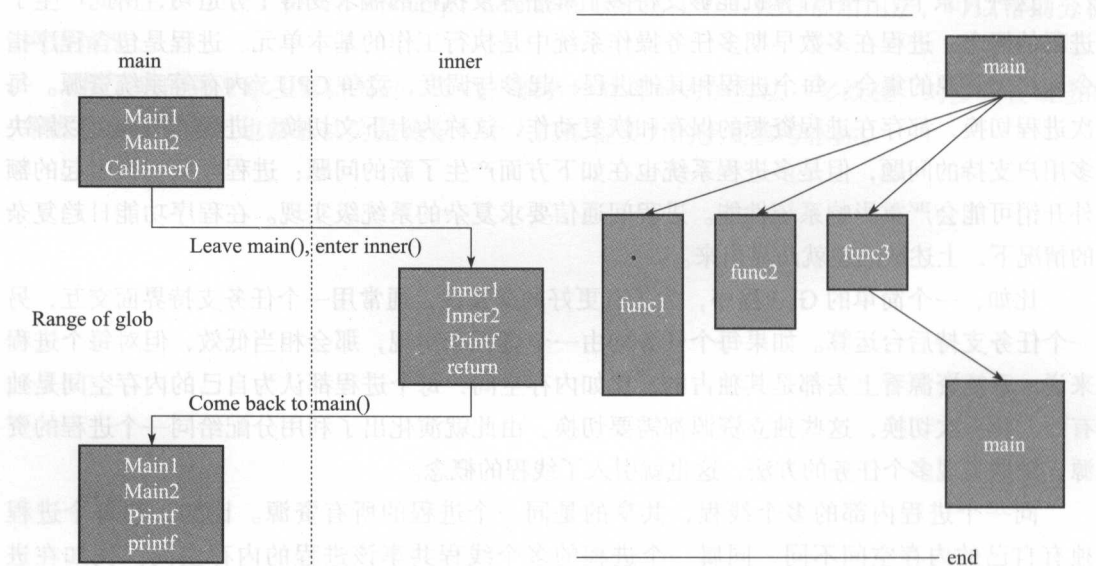


图 9-1 函数执行状态

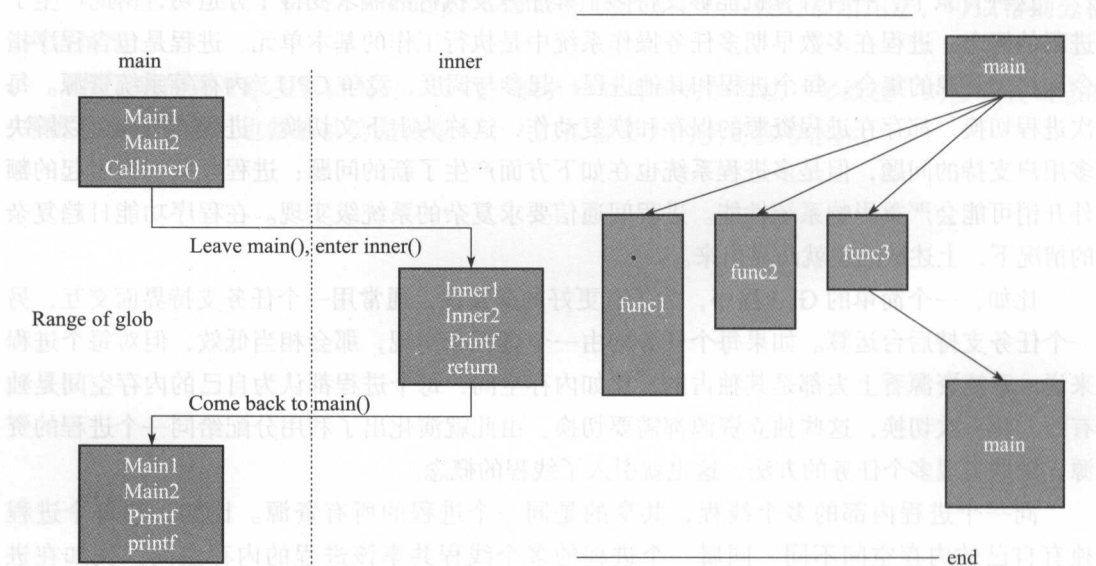


图 9-2 多线程流程示意图

main() 到 func3() 再到 main() 构成一个线程, 此外 func1() 和 func2() 构成另外两个线程。操作系统一般都有一些系统调用来让一个函数运行成为一个新的线程。

回忆下栈的功能和用途: 一个栈中只有最下方的帧可被读写, 相应的, 也只有该帧对应的那个函数被激活, 处于工作状态。为了实现多线程, 则必须绕开栈的限制。为此, 在创建一个新的线程时, 需要为这个线程建一个新的栈, 每个栈对应一个线程。当某个栈执行到全部弹出时, 对应线程完成任务, 并结束。所以, 多线程的进程在内存中有多个栈, 多个栈之间以一定的空白区域隔开, 以备栈的增长。每个线程可调用自己栈最下方的帧中的参数和变量, 并与其他线程共享内存中的 Text、heap 和 global data 区域。对应上面图 9-2 的例子, 这里的进程空间中需要有 3 个栈。

要注意的是, 对于多线程来说, 由于同一个进程空间中存在多个栈, 任何一个空白区域被填满都会导致栈溢出。

这就是多线程, 与栈密切相关。

9.2 多线程的创建与结束

1. 线程的创建

多线程函数需要包括的头文件:

```
#include<pthread.h>
```

线程创建函数: pthread_create, 它的函数原型是:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void
*(start_routine) (void *), void *arg);
```

这里看到一种新的数据类型 pthread_t, pthread_t 的定义是 typedef unsigned long int pthread_t。也就是说, pthread_t 其实是 unsigned long int 来的, 是个无符号的长整型。

pthread_create 函数第一个参数为指向线程标识符的指针, 第二个参数用来设置线程属性, 第三个参数是线程运行函数的起始地址, 最后一个参数是运行函数的参数。

pthread_create 函数的返回值: 若线程创建成功, 则返回 0; 若线程创建失败, 则返回出错编号, 并且 *thread 中的内容是未定义的。返回成功时, 由 thread 指向的内存单元将被设置为新建线程的线程 ID。attr 参数用于指定各种不同的线程属性。新创建的线程从 start_routine 函数的地址开始运行, 该函数只有一个万能指针参数 arg, 如果需要向 start_routine 函数传递的参数不止一个, 那么需要把这些参数放到同一个结构中, 然后把这个结构的地址作为 arg 的参数传入。

pthread_join 函数用来等待一个线程的结束, 其函数原型为:

```
int pthread_join (pthread_t thread, void **retval);
```


第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

pthread_exit 函数，一个线程的结束有两种途径：①函数已经结束，调用它的线程也就结束了；②通过函数 **pthread_exit** 来实现。它的函数原型为：

```
void pthread_exit (void *retval);
```

其中，唯一的参数是函数的返回代码。

pthread_join 和 **pthread_exit** 的区别如下所述。

(1) **pthread_join** 一般是主线程来调用，用来等待子线程退出，因为是等待，所以是阻塞的，一般主线程会依次添加所有它创建的子线程。

(2) **pthread_exit** 一般是子线程调用，用来结束当前线程。

(3) 子线程可以通过 **pthread_exit** 传递一个返回值，而主线程通过 **pthread_join** 获得该返回值，从而判断该子线程的退出是正常还是异常。

下面例 9.1 讲述了线程的创建与结束。

【例 9.1】 线程的创建与结束。

```
#include <stdio.h>
#include <pthread.h>
void* say_hello(void* args){
    /* 线程的运行函数，必须 void*，没说的表示返回通用指针、输入通用指针 */
    printf("hello from thread\n");
    pthread_exit((void*)1);
}
int main(){
    pthread_t tid;
    int iRet = pthread_create(&tid, NULL, say_hello, NULL);
    /* 参数依次是：创建的线程 id，线程参数，调用函数名，传入的函数参数 */
    if (iRet){
        printf("pthread_create error: iRet=%d\n", iRet);
        return iRet;
    }
    void *retval;
    iRet=pthread_join(tid, &retval);
    if (iRet){
        printf("pthread_join error: iRet=%d\n", iRet);
        return iRet;
    }
    printf("retval=%ld\n", (long)retval);
    return 0;
}
```

编译程序的命令是：**g++ -lpthread -o test test.cpp**，这里需要连接静态库文件 **pthread**。执行 **./test** 命令可得到：


```
hello from thread
temp=1
```

例 9.1 中，say_hello 函数是线程的运行函数，返回值是 void* 类型，也就是返回通用指针、输入通用指针。在线程函数中调用 pthread_exit，以便返回值被主线程接收，如下所示。

```
void* say_hello(void* args){
    /* 线程的运行函数，必须 void*，没说的表示返回通用指针、输入通用指针 */
    printf("hello from thread\n");
    pthread_exit((void*)1);
}
```

创建线程，参数依次是：创建的线程 id、线程参数、调用函数名、传入的函数参数。这里，线程的 id 放在变量 tid 里，线程参数和函数参数都为空，函数名是 say_hello。

```
int iRet = pthread_create(&tid, NULL, say_hello, NULL);
```

调用 pthread_join 函数，获取线程的返回值。注意，由于本程序是在 64 位机器上执行的，所以指针类型和 long 类型大小相等，都是 8Byte，并且将 temp 强制转换成了 long 类型。如果强制转换成 int 类型，编译时会提示 “error: cast from 'void*' to 'int' loses precision”，即丢失精度。

```
void *retval;
iRet=pthread_join(tid,&retval);
if (iRet){
    printf("pthread_join error: iRet=%d\n",iRet);
    return iRet;
}
printf("retval=%ld\n",(long)retval);
```

在例 9.1 的基础上，如果线程调用的函数是在一个类中时，应该把该函数写成静态成员函数，如例 9.2 所示。

【例 9.2】创建线程时传入类的成员函数。

```
#include <stdio.h>
#include <pthread.h>
class Hello{
public:
    static void* say_hello(void* args){
        /* 线程的运行函数，必须 void*，没说的表示返回通用指针、输入通用指针 */
        printf("hello from thread\n");
        pthread_exit((void*)1);
    }
};
int main(){
    pthread_t tid;
    int iRet = pthread_create(&tid, NULL, Hello::say_hello, NULL);
    /* 参数依次是：创建的线程 id，线程参数，调用函数名，传入的函数参数 */
```

```

    if (iRet){
        printf("pthread_create error: iRet=%d\n",iRet);
        return iRet;
    }
    void *retval;
    iRet=pthread_join(tid,&retval);
    if (iRet){
        printf("pthread_join error: iRet=%d\n",iRet);
        return iRet;
    }
    printf("retval=%ld\n", (long)retval);
    return 0;
}

```

程序的执行结果是：

```

hello from thread
temp=1

```

例 9.2 与例 9.1 相比，就是把函数 say_hello 换成了类中的静态成员函数，程序的输出结果是一样的。

2. 向线程传递参数

线程创建完成后，那如何在线程调用函数时传入参数呢？如例 9.3 所示。

【例 9.3】 在线程调用函数时传入 int 类型的参数。

```

#include <stdio.h>
#include <pthread.h>
void* say_hello(void* args){
    int i=*(int*)args;
    printf("hello from thread,i=%d\n",i);
    pthread_exit((void*)1);
}
int main(){
    pthread_t tid;
    int para=10;
    int iRet = pthread_create(&tid, NULL, say_hello, &para);
    if (iRet){
        printf("pthread_create error: iRet=%d\n",iRet);
        return iRet;
    }
    void *retval;
    iRet=pthread_join(tid,&retval);
    if (iRet){
        printf("pthread_join error: iRet=%d\n",iRet);
        return iRet;
    }
    printf("retval=%ld\n", (long)retval);
    return 0;
}

```

程序的执行结果:

```
hello from thread,i=10
retval=1
```

例 9.3 中, 给线程调用的函数传递了一个 int 类型的参数 para, 值是 10, 代码如下所示。

```
int para=10;
int iRet = pthread_create(&tid, NULL, say_hello, &para);
if (iRet){
    printf("pthread_create error: iRet=%d\n",iRet);
    return iRet;
}
```

并在函数中把参数给打印出来, 代码如下所示。

```
void* say_hello(void* args){
    int i=*(int*)args;
    printf("hello from thread,i=%d\n",i);
    pthread_exit((void*)1);
}
```

pthread_create (&tid,&attr,&func, (void) arg); 看起来只能传递一个参数给 func, 如果要传一个以上的参数呢, 则应该把它们放到一个结构体里, 然后传递这个结构体, 可参见例 9.4。

【例 9.4】 在线程调用函数时传入一个结构体。

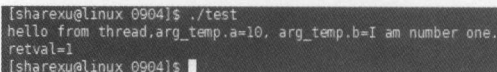
```
#include <stdio.h>
#include <pthread.h>
#include <string.h>
struct arg_type{
    int a;
    char b[100];
};
void* say_hello(void* args){
    arg_type arg_temp=*(arg_type*)args;
    printf("hello from thread,arg_temp.a=%d, arg_temp.b=%s\n",arg_temp.a,arg_
temp.b);
    pthread_exit((void*)1);
}
int main(){
    pthread_t tid;
    arg_type arg_temp;
    arg_temp.a=10;
    char temp[100]="I am number one.";
    strncpy(arg_temp.b,temp,sizeof(temp));
    int iRet = pthread_create(&tid, NULL, say_hello, &arg_temp);
    if (iRet){
        printf("pthread_create error: iRet=%d\n",iRet);
        return iRet;
    }
```

```

    }
    void *retval;
    iRet=pthread_join(tid,&retval);
    if (iRet){
        printf("pthread_join error: iRet=%d\n",iRet);
        return iRet;
    }
    printf("retval=%ld\n", (long)retval);
    return 0;
}

```

程序的执行结果如图 9-3 所示。



```

[sharexu@linux 0904]$ ./test
hello from thread,arg_temp.a=10, arg_temp.b=I am number one.
retval=1
[sharexu@linux 0904]$

```

图 9-3 例 9.4 中程序的执行结果

例 9.4 中，向 `pthread_create` 中调用的函数 `say_hello` 传入了一个结构体，传入时直接引用结构体的地址即可，代码如下所示。

```

arg_type arg_temp;
arg_temp.a=10;
char temp[100]="I am number one.";
strncpy(arg_temp.b,temp,sizeof(temp));
int iRet = pthread_create(&tid, NULL, say_hello, &arg_temp);

```

3. 获得线程 id

我们有两种方式可以打印线程的 id：①在线程调用函数中使用 `pthread_self` 函数来获得线程 id；②在创建函数时生成的 id。我们来分别实现一下这两个方法，来判断结果是否一致，参见例 9.6。

【例 9.6】获得线程的 id。

```

#include<stdio.h>
#include<pthread.h>
void *function(void *arg){
    printf("thread id in pthread=%lu\n", pthread_self());
    pthread_exit((void *)1);
}

int main(){
    int i=10;
    pthread_t thread;
    int iRet = pthread_create(&thread, NULL, &function, &i);
    if(iRet){
        printf("pthread_create error, iRet=%d\n",iRet);
        return iRet;
    }
    printf("thread id in process=%lu\n", thread);
    void *retval;
    iRet=pthread_join(thread, &retval);
    if(iRet){
        printf("pthread_join error, iRet=%d\n",iRet);
        return iRet;
    }
}

```

```

    }
    printf("retval=%ld\n", (long*)retval);
    return 0;
}

```

程序的执行结果如图 9-4 所示。

可见，用 `pthread_create` 创建的线程 id 和用 `pthread_self` 函数获得的线程 id 是一样的。

```

[sharexu@linux 0906]$ ./test
thread id in process=139906278786816
thread id in pthread=139906278786816
retval=1

```

图 9-4 例 9.6 中程序的执行结果

9.3 线程的属性

线程有一组属性是可以在线程被创建时指定的。该组属性被封装在一个对象中，该对象可用来设置一个或一组线程的属性。线程属性对象的类型为 `pthread_attr_t`。 `pthread_attr_t` 包含在 `pthread.h` 头文件中。

线程属性结构如下所示。

```

typedef struct
{
    int          detachstate;           // 线程的分离状态
    int          schedpolicy;          // 线程调度策略
    struct sched_param schedparam;      // 线程的调度参数
    int          inheritsched;         // 线程的继承性
    int          scope;                // 线程的作用域
    size_t       guardsize;            // 线程栈末尾的警戒缓冲区大小
    int          stackaddr_set;        // 线程的栈设置
    void*        stackaddr;           // 线程栈的位置
    size_t       stacksize;           // 线程栈的大小
} pthread_attr_t;

```

属性值不能直接设置，必须使用相关函数进行操作，初始化的函数为 `pthread_attr_init`，且这个函数必须在 `pthread_create` 函数之前调用，之后必须用 `pthread_attr_destroy` 函数来释放资源。线程属性主要包括如下属性：作用域（scope）、栈尺寸（stack size）、栈地址（stack address）、优先级（priority）、分离的状态（detached state）、调度策略和参数（scheduling policy and parameters）等。默认的属性为非绑定、非分离、默认 1MB 大小的堆栈、与父进程同样级别的优先级。

POSIX.1 指定了一系列方法获取和设置 `pthread_attr_t` 结构里面的各个属性，如下所述。

（1）分离状态（detached state）：若线程终止时，线程处于分离状态，系统将不保留线程终止的状态；当不需要线程的终止状态时，可以分离线程（调用 `pthread_detach` 函数）；若在线程创建的时候，就已经知道以后不需要使用线程的终止状态，可以在线程创建属性里面指定该状态，那么线程一开始就处于分离状态。通过下面两个函数，设置和获取线程的分离属性：

```
int pthread_attr_getdetachstate ( const pthread_attr_t *attr, int *state );
int pthread_attr_setdetachstate ( pthread_attr_t *attr, int state );
```

该属性的可选值有：PTHREAD_CREATE_DETACHED、PTHREAD_CREATE_JOINABLE。

(2) 栈地址 (stack address)：POSIX.1 定义了两个常量 POSIX_THREAD_ATTR_STACKADDR 和 POSIX_THREAD_ATTR_STACKSIZE 以检测系统是否支持栈属性。当然也可以给 sysconf 函数传递 _SC_THREAD_ATTR_STACKADDR 或 _SC_THREAD_ATTR_STACKSIZE 来进行检测。当进程栈地址空间不够用时，指定新建线程使用由 malloc 分配的空间作为自己的栈空间。通过 pthread_attr_setstackaddr 和 pthread_attr_getstackaddr 两个函数分别设置和获取线程的栈地址。传给 pthread_attr_setstackaddr 函数的地址是缓冲区的低地址 (不一定是栈的开始地址，栈可能从高地址往低地址增长)，代码如下所示。

```
int pthread_attr_getstackaddr ( const pthread_attr_t *attr, void **addr );
int pthread_attr_setstackaddr ( pthread_attr_t *attr, void *addr );
```

(3) 栈大小 (stack size)：当系统中有很多线程时，可能需要减小每个线程栈的默认大小，防止进程的地址空间不够用；当线程调用的函数会分配很大的局部变量或者函数调用层次很深时，可能需要增大线程栈的默认大小。函数 pthread_attr_getstacksize 和 pthread_attr_setstacksize 被提供来解决这个问题。

```
int pthread_attr_getstacksize ( const pthread_attr_t *attr, size_t *size );
int pthread_attr_setstacksize ( pthread_attr_t *attr, size_t size );
```

函数 pthread_attr_getstack 和 pthread_attr_setstack 函数可以同时操作栈地址和栈大小两个属性，代码如下所示。

```
int pthread_attr_getstack ( const pthread_attr_t *attr, void **stackaddr, size_t *size );
int pthread_attr_setstack ( pthread_attr_t *attr, void *stackaddr, size_t size );
```

(4) 栈保护区大小 (stack guard size)：在线程栈顶留出一段空间，防止栈溢出。当栈指针进入这段保护区时，系统会发出错误提示，通常会发送信号给线程。该属性默认值是 PAGESIZE 的大小，该属性被设置时，系统会自动将该属性大小补齐为页大小的整数倍。当改变栈地址属性时，栈保护区大小通常清零。

```
int pthread_attr_getguardsize ( const pthread_attr_t *attr, size_t *guardsize );
int pthread_attr_setguardsize ( pthread_attr_t *attr, size_t guardsize );
```

(5) 线程优先级 (priority)：新线程的优先级为 0。

```
int pthread_attr_getschedparam (const pthread_attr_t *restrict attr, struct
sched_param *restrict param);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr, const struct
sched_param *restrict param);
```


(6) 继承父进程优先级 (inheritsched): 新线程不继承父线程调度优先级。

(7) 调度策略 (schedpolicy): 新线程使用 SCHED_OTHER 调度策略。线程一旦开始运行, 直到被抢占或者直到线程阻塞或停止为止。

```
int pthread_attr_setschedpolicy(pthread_attr_t* attr, int policy)
int pthread_attr_setschedparam (pthread_attr_t* attr, struct sched_param*
param);
```

(8) 争用范围 (scope): 建立线程的争用范围 (PTHREAD_SCOPE_SYSTEM 或 PTHREAD_SCOPE_PROCESS)。使用 PTHREAD_SCOPE_SYSTEM 时, 此线程将与系统中的所有线程进行竞争。使用 PTHREAD_SCOPE_PROCESS 时, 此线程将与进程中的其他线程进行竞争, 又称为绑定状态, PTHREAD_SCOPE_SYSTEM (绑定的) 和 PTHREAD_SCOPE_PROCESS (非绑定的)。具有不同范围状态的线程可以在同一个系统甚至同一个进程中共存。进程范围只允许这种线程与同一进程中的其他线程争用资源, 而系统范围则允许此类线程与系统内的其他所有线程争用资源。实际上, 从 Solaris 9 发行版开始, 系统就不再区分这两个范围。

```
int pthread_attr_getscope(const pthread_attr_t *restrict attr, int *restrict
contentionscope);
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

(9) 线程并行级别 (concurrency) 应用程序使用 pthread_setconcurrency() 通知系统所需的并发级别。

```
int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);
```

POSIX 标准指定了 3 种调度策略: 先入先出策略 (SCHED_FIFO)、循环策略 (SCHED_RR) 和自定义策略 (SCHED_OTHER)。SCHED_FIFO 是基于队列的调度程序, 对于每个优先级都会使用不同的队列。SCHED_RR 与 FIFO 相似, 不同的是前者的每个线程都有一个执行时间配额。

SCHED_FIFO: 如果调用进程具有有效的用户 ID 为 0, 则争用范围为系统 (PTHREAD_SCOPE_SYSTEM) 的先入先出线程属于实时 (RT) 调度类。如果这些线程未被优先级更高的线程抢占, 则会继续处理该线程, 直到该线程放弃或阻塞为止。对于具有进程争用范围 (PTHREAD_SCOPE_PROCESS) 的线程或其调用进程没有有效用户 ID 为 0 的线程, 请使用 SCHED_FIFO。SCHED_FIFO 基于 TS 调度类。

SCHED_RR: 如果调用进程具有有效的用户 ID 0, 则争用范围为系统 (PTHREAD_SCOPE_SYSTEM) 的循环线程, 属于实时 (RT) 调度类。如果这些线程未被优先级更高的线程抢占, 并且这些线程没有放弃或阻塞, 则在系统确定的时间段内将一直执行这些线程。对于具有进程争用范围 (PTHREAD_SCOPE_PROCESS) 的线程, 请使用 SCHED_RR (基于 TS 调度类)。此外, 这些线程的调用进程没有有效的用户 ID 0。

将线程设置为结束状态分离后，线程的结束状态将不能被进程中的其他线程得到，同时保存线程结束状态的存储区域也将变得不能应用。下面的例 9.7 演示了如何分离出一个线程。

【例 9.7】分离一个线程。

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
void * tfnl(void * arg){
    printf("the thread\n");
    return NULL;
}
int main(void){
    int iRet;
    pthread_t tid;
    pthread_attr_t attr;
    iRet = pthread_attr_init(&attr);
    if(iRet){
        printf("can't init attr %s/n", strerror(iRet));
        return iRet;
    }
    iRet = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if(iRet){
        printf("can't set attr %s\n", strerror(iRet));
        return iRet;
    }
    iRet = pthread_create(&tid, &attr, tfnl, NULL);
    if(iRet){
        printf("can't create thread %s\n", strerror(iRet));
        return iRet;
    }
    iRet = pthread_join(tid, NULL);
    if(iRet){
        printf("thread has been detached\n");
        return iRet;
    }
    return 0;
}
```



```
[sharexu@linux 0907]$ ./test
thread has been detached
```

程序的执行结果如图 9-5 所示。

图 9-5 例 9.7 中程序的执行结果

例 9.7 中，把线程的属性设置为了线程与线程结束状态分离，代码如下：

```
iRet = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
```

创建一个线程时，带上这个属性，代码如下：

```
iRet = pthread_create(&tid, &attr, tfnl, NULL);
```

由于状态分离，因此得不到线程的结束状态信息，pthread_join 函数会出错，代码如下：

```

iRet = pthread_join(tid, NULL);
if(iRet){
    printf("thread has been detached\n");
    return iRet;
}

```

由程序的执行结果显示，该线程已成功分离。

例 9.7 中我们是在线程创建之前，就将它的属性设置为分离状态。下面的例 9.8 演示了如何分离一个已经创建的线程。

【例 9.8】分离一个已经创建的线程。

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
void * tfn1(void * arg){
    printf("the sub thread sleeping for 5 seconds\n");
    sleep(5); /* 休眠 5 秒，等待主线程将该线程设置为分离状态 */
    printf("the thread done\n");
    return NULL;
}
int main(void){
    int iRet;
    pthread_t tid;
    iRet = pthread_create(&tid, NULL, tfn1, NULL);
    /* 创建一个线程，这个线程和结束状态是分离的 */
    if(iRet){
        printf("can't create thread %s\n", strerror(iRet));
        return iRet;
    }
    iRet = pthread_detach(tid); /* 设置线程为分离状态 */
    if(iRet){
        printf("can't detach thread %s\n", strerror(iRet));
        return iRet;
    }
    iRet = pthread_join(tid, NULL);
    /* 由于状态分离，因此得不到线程的结束状态信息，此函数会出错 */
    if(iRet){
        printf("thread has been detached\n");
    }
    printf("the main thread sleeping for 8 seconds\n");
    sleep(8);
    printf("the main thread done.\n");
    return 0;
}

```

程序的执行结果如图 9-6 所示。

例 9.8 中，用 pthread_detach 函数将一个已创建的线程

```

[sharexu@linux 0908]$ ./test
thread has been detached
the main thread sleeping for 8 seconds
the sub thread sleeping for 5 seconds
the thread done
the main thread done.
[sharexu@linux 0908]$

```

图 9-6 例 9.8 中程序的执行结果

设置为分离状态，导致用 `pthread_join` 函数获取不到它的结束状态信息，代码如下：

```
iRet = pthread_detach(tid);
```

9.4 多线程同步

多线程相当于一个并发系统，一般同时执行多个任务。如果多个任务可以共享资源，特别是同时写入某个变量的时候，就需要解决同步的问题。这里用多线程来模拟火车售票系统，具体代码见例 9.9。

1. 多线程同步问题

【例 9.9】用多线程来模拟火车售票系统。

```
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
int total_ticket_num=20;
void *sell_ticket(void *arg){
    for(int i=0;i<20;i++){
        if(total_ticket_num>0){
            sleep(1);
            printf("sell the %dth ticket\n",20-total_ticket_num+1);
            total_ticket_num--;
        }
    }
    return 0;
}

int main(){
    int iRet;
    pthread_t tids[4];
    int i=0;
    for(i=0;i<4;i++){
        int iRet = pthread_create(&tids[i], NULL, &sell_ticket, NULL);
        if(iRet){
            printf("pthread_create error, iRet=%d\n",iRet);
            return iRet;
        }
    }
    sleep(20);
    void *retval;
    for(i=0;i<4;i++){
        iRet=pthread_join(tids[i], &retval);
        if(iRet){
            printf("tid=%d join error, iRet=%d\n",tids[i],iRet);
            return iRet;
        }
    }
    printf("retval=%ld\n",(long*)retval);
```

```

    }
    return 0;
}

```

程序的执行结果如图 9-7 所示。

程序的执行结果并不如预期所想，总票数只有 20 张，却卖掉了 23 张。程序创建了 4 个线程，每个线程都执行卖票函数。总票数是存放在一个全局变量 `total_ticket_num` 里的，代码如下：

```

void *sell_ticket(void *arg){
    for(int i=0;i<20;i++){
        if(total_ticket_num>0){
            sleep(1);
            printf("sell the %dth ticket\
n",20-total_ticket_num+1);
            total_ticket_num--;
        }
    }
    return 0;
}

```

```

[sharexu@linux 0909]$ g++ -lpthread -o test test.cpp
[sharexu@linux 0909]$ ./test
sell the 1th ticket
sell the 2th ticket
sell the 3th ticket
sell the 4th ticket
sell the 5th ticket
sell the 6th ticket
sell the 7th ticket
sell the 8th ticket
sell the 9th ticket
sell the 10th ticket
sell the 11th ticket
sell the 12th ticket
sell the 13th ticket
sell the 14th ticket
sell the 15th ticket
sell the 16th ticket
sell the 17th ticket
sell the 18th ticket
sell the 19th ticket
sell the 20th ticket
sell the 21th ticket
sell the 22th ticket
sell the 23th ticket
retval=0
retval=0
retval=0
retval=0
[sharexu@linux 0909]$

```

图 9-7 例 9.9 中程序的执行结果

事实上，如果只有一个线程执行上面的程序的时候（相当于一个窗口售票），则没有问题。但如果多个线程都执行上面的程序（相当于多个窗口售票），就会出现問題。其根本原因在于同时发生的各个线程都可以对 `total_ticket_num` 读取和写入。

这里的 `if` 结构会判断是否有剩余的票（`total_ticket_num > 0`），如果有则卖票（`total_ticket_num = total_ticket_num - 1`）。某个线程会先判断是否有票（比如说此时 `total_ticket_num` 为 1），但两个指令之间存在一个时间窗口，其他线程可能在此时间窗口内执行卖票操作（`total_ticket_num = total_ticket_num - 1`），导致该线程卖票的条件不再成立。但该线程由于已经执行过了判断指令，所以无从知道 `total_ticket_num` 发生了变化，所以继续执行卖票指令，以至于卖出不存在的票（`total_ticket_num` 成为负数）。对于一个真实的售票系统来说，这将成为一个严重的错误（售出了过多的票，火车超员）。

在并发情况下，指令执行的先后顺序由内核决定。同一个线程内部，指令按照先后顺序执行，但不同线程之间的指令很难说清楚哪一个会先执行。如果运行的结果依赖于不同线程执行的先后的话，那么就会造成竞争条件，在这样的状况下，计算机的结果很难预知，所以应该尽量避免竞争条件的形成。最常见的解决竞争条件的方法是将原先分离的两个指令构成不可分割的一个原子操作，而其他任务不能插入到原子操作中。

对于多线程程序来说，同步是指在一定的时间内只允许某一个线程访问某个资源。而在此时间内，不允许其他的线程访问该资源。可以通过互斥锁（`mutex`）、条件变量（`condition variable`）、读写锁（`reader-writer lock`）和信号量（`semaphore`）来同步资源。

2. 互斥锁

互斥锁是一个特殊的变量，它有锁上（lock）和打开（unlock）两个状态。互斥锁一般被设置成全局变量。打开的互斥锁可以由某个线程获得。一旦获得，这个互斥锁会锁上，此后只有该线程有权打开，其他想要获得互斥锁的线程，会等待直到互斥锁再次打开的时候。我们可以将互斥锁想象成一个只能容纳一个人的洗手间，当某个人进入洗手间的时候，可以从里面将洗手间锁上，其他人只能在互斥锁外面等待那个人出来，才能进去。但在外面等候的人并没有排队，谁先看到洗手间空了，就可以首先冲进去。

上面的问题很容易使用互斥锁的问题解决，程序可以改为这样，如例 9.10 所示。

【例 9.10】用互斥锁同步资源。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

pthread_mutex_t mutex_x= PTHREAD_MUTEX_INITIALIZER;
int total_ticket_num=20;

void *sell_ticket(void *arg){
    for(int i=0;i<20;i++){
        pthread_mutex_lock(&mutex_x);
        if(total_ticket_num>0){
            sleep(1);
            printf("sell the %dth ticket\n",20-total_ticket_num+1);
            total_ticket_num--;
        }
        pthread_mutex_unlock(&mutex_x);
    }
    return 0;
}

int main(){
    int iRet;
    pthread_t tids[4];
    int i=0;
    for(i=0;i<4;i++){
        int iRet = pthread_create(&tids[i], NULL, &sell_ticket, NULL);
        if(iRet){
            printf("pthread_create error, iRet=%d\n",iRet);
            return iRet;
        }
    }
    sleep(30);
    void *retval;
    for(i=0;i<4;i++){
        iRet=pthread_join(tids[i], &retval);
        if(iRet){
            printf("tid=%d join error, iRet=%d\n",tids[i],iRet);
        }
    }
}
```



```

        return iRet;
    }
    printf("retval=%ld\n", (long*)retval);
}
return 0;
}

```

程序的执行结果如图 9-8 所示。

程序执行结果符合预期，一共 20 张票，也只卖出了 20 张。例 9.10 与例 9.9 的不同之处只是加了全局的互斥锁，并且在线程执行的函数 `sell_ticket` 中，for 循环每次对全局变量 `total_ticket_num` 操作前加锁，操作后解锁。

```

[sharexu@linux 0910]$ ./test
sell the 1th ticket
sell the 2th ticket
sell the 3th ticket
sell the 4th ticket
sell the 5th ticket
sell the 6th ticket
sell the 7th ticket
sell the 8th ticket
sell the 9th ticket
sell the 10th ticket
sell the 11th ticket
sell the 12th ticket
sell the 13th ticket
sell the 14th ticket
sell the 15th ticket
sell the 16th ticket
sell the 17th ticket
sell the 18th ticket
sell the 19th ticket
sell the 20th ticket
retval=0
retval=0
retval=0
retval=0

```

图 9-8 例 9.10 中程序的执行结果

```

pthread_mutex_t mutex_x= PTHREAD_MUTEX_
INITIALIZER;
int total_ticket_num=20;
void *sell_ticket(void *arg){
    for(int i=0;i<20;i++){
        pthread_mutex_lock(&mutex_x);
        if(total_ticket_num>0){
            sleep(1);
            printf("sell the %dth ticket\n",20-total_ticket_num+1);
            total_ticket_num--;
        }
        pthread_mutex_unlock(&mutex_x);
    }
    return 0;
}

```

第一个执行 `pthread_mutex_lock()` 的线程会先获得 `mutex_x`，其他想要获得 `mutex_x` 的线程必须等待，直到第一个线程执行到 `pthread_mutex_unlock()` 释放 `mutex_x`，才可以获得 `mutex_x`，并继续执行线程。所以线程在 `pthread_mutex_lock()` 和 `pthread_mutex_unlock()` 之间操作时，不会被其他线程影响，就构成了一个原子操作。

需要注意的时候，如果存在某个线程依然使用原先的程序，即不尝试获得 `mutex_x`，而直接修改 `total_ticket_num`，互斥锁不能阻止该程序修改 `total_ticket_num`，互斥锁就失去了保护资源的意义。所以，互斥锁机制需要程序员自己来写出完善的程序来实现互斥锁的功能。

互斥锁的使用过程中，主要有 `pthread_mutex_init`、`pthread_mutex_destory`、`pthread_mutex_lock` 和 `pthread_mutex_unlock` 这几个函数，分别完成锁的初始化、锁的销毁、上锁和释放锁操作。锁的创建有两种方式，静态和动态。例 9.10 中是以静态方式创建了锁，代码如下：

```
pthread_mutex_t mutex_x= PTHREAD_MUTEX_INITIALIZER;
```

另外锁可以用 `pthread_mutex_init` 函数动态地创建，函数原型如下：

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

对锁的操作主要包括加锁 `pthread_mutex_lock()`、解锁 `pthread_mutex_unlock()` 和测试加锁 `pthread_mutex_trylock()` 3 个，代码如下：

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

`pthread_mutex_trylock()` 语义与 `pthread_mutex_lock()` 类似，不同的是在锁已经被占据时返回 `EBUSY`，而不是挂起等待。

【例 9.11】使用 `pthread_mutex_trylock` 测试加锁。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

pthread_mutex_t mutex_x= PTHREAD_MUTEX_INITIALIZER;
int total_ticket_num=20;
void *sell_ticket1(void *arg){
    for(int i=0;i<20;i++){
        pthread_mutex_lock(&mutex_x);
        if(total_ticket_num>0){
            printf("thread1 sell the %dth ticket\n",20-total_ticket_num+1);
            total_ticket_num--;
        }
        sleep(1);
        pthread_mutex_unlock(&mutex_x);
        sleep(1);
    }
    return 0;
}

void *sell_ticket2(void *arg){
    int iRet=0;
    for(int i=0;i<10;i++){
        iRet=pthread_mutex_trylock(&mutex_x);
        if(iRet==EBUSY){
            printf ("sell_ticket2:the variable is locked by sell_ticket1.\n");
        }else if(iRet==0){
            if(total_ticket_num>0){
                printf("thread2 sell the %dth ticket\n",20-total_ticket_num+1);
                total_ticket_num--;
            }
            pthread_mutex_unlock(&mutex_x);
        }
    }
}
```

```

    }
    sleep(1);
}
return 0;
}

int main(){
    pthread_t tids[2];
    int iRet = pthread_create(&tids[0], NULL, &sell_ticket1, NULL);
    if(iRet){
        printf("pthread_create error, iRet=%d\n",iRet);
        return iRet;
    }
    iRet = pthread_create(&tids[1], NULL, &sell_ticket2, NULL);
    if(iRet){
        printf("pthread_create error, iRet=%d\n",iRet);
        return iRet;
    }
    sleep(30);
    void *retval;
    iRet=pthread_join(tids[0], &retval);
    if(iRet){
        printf("tid=%d join error, iRet=%d\n",tids[0],iRet);
    }else{
        printf("retval=%ld\n", (long*)retval);
    }
    iRet=pthread_join(tids[1], &retval);
    if(iRet){
        printf("tid=%d join error, iRet=%d\n",tids[1],iRet);
    }else{
        printf("retval=%ld\n", (long*)retval);
    }
    return 0;
}

```

程序的执行结果如图 9-9 所示。

这里设计了一个函数 `sell_ticket1`，直接使用 `pthread_mutex_lock` 和 `pthread_mutex_unlock` 来加锁和解锁；而函数 `sell_ticket2` 则是用 `pthread_mutex_trylock` 来进行尝试加锁。如果尝试失败，会返回 `EBUSY`。这样就可以更加清晰地看到两个线程争夺资源的现象。

3. 条件变量

互斥量是线程程序必需的工具，但并非是万能的。例如，如果线程正在等待共享数据内某个条件出现，那会发生什么呢？它可能重复对互斥对象锁定和

```

[sharexu@linux 0911]$ g++ -lpthread -o test test.cpp
[sharexu@linux 0911]$ ./test
thread2 sell the 1th ticket
thread1 sell the 2th ticket
sell_ticket2:the variable is locked by sell_ticket1.
thread2 sell the 3th ticket
thread1 sell the 4th ticket
sell_ticket2:the variable is locked by sell_ticket1.
thread2 sell the 5th ticket
thread1 sell the 6th ticket
sell_ticket2:the variable is locked by sell_ticket1.
thread2 sell the 7th ticket
thread1 sell the 8th ticket
sell_ticket2:the variable is locked by sell_ticket1.
thread2 sell the 9th ticket
thread1 sell the 10th ticket
sell_ticket2:the variable is locked by sell_ticket1.
thread1 sell the 11th ticket
thread1 sell the 12th ticket
thread1 sell the 13th ticket
thread1 sell the 14th ticket
thread1 sell the 15th ticket
thread1 sell the 16th ticket
thread1 sell the 17th ticket
thread1 sell the 18th ticket
thread1 sell the 19th ticket
thread1 sell the 20th ticket
retval=0
retval=0

```

图 9-9 例 9.11 中程序的执行结果

解锁，每次都会检查共享数据结构，以查找某个值。但这是在浪费时间和资源，而且这种繁忙查询的效率非常低。

在每次检查之间，可以让调用线程短暂地进入睡眠，比如睡眠 3 秒，但是由此线程代码就无法最快作出响应。真正需要的是这样一种方法：当线程在等待满足某些条件时使线程进入睡眠状态，一旦条件满足，就唤醒因等待满足特定条件而睡眠的线程。如果能够做到这一点，线程代码将是非常高效的，并且不会占用宝贵的互斥对象锁。而这正是条件变量能做的事！

条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补互斥锁的不足，它常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其他的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程，这些线程将重新锁定互斥锁并重新测试条件是否满足。

这里先介绍下条件变量的相关函数使用。

(1) 创建：条件变量和互斥锁一样，都有静态、动态两种创建方式。

静态方式使用 `PTHREAD_COND_INITIALIZER` 常量，函数原型是：

```
pthread_cond_t cond=PTHREAD_COND_INITIALIZER
```

动态方式则使用 `pthread_cond_init` 函数，`pthread_cond_init` 的函数原型是：

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr)
```

使用 `cond_attr` 指定的属性初始化条件变量 `cond`，当 `cond_attr` 为 `NULL` 时，使用默认的属性。`LinuxThreads` 实现条件变量不支持属性，因此 `cond_attr` 参数实际被忽略。

(2) 注销：注销一个条件变量需要调用 `pthread_cond_destroy()`，它的函数原型是：

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

只有在没有线程在该条件变量上等待的时候才能注销这个条件变量，否则返回 `EBUSY`。因为 `Linux` 实现的条件变量没有分配什么资源，所以注销动作只包括检查是否有等待线程。

(3) 等待：等待条件有两种方式——条件等待 `pthread_cond_wait()` 和计时等待 `pthread_cond_timedwait()`。其中计时等待方式如果在给定时刻前条件没有满足，则返回 `ETIMEDOUT`，结束等待。其中 `abstime` 以与 `time()` 系统调用相同意义的绝对时间形式出现，0 表示格林尼治时间 1970 年 1 月 1 日 0 时 0 分 0 秒。

无论哪种等待方式，都必须和一个互斥锁配合，以防止多个线程同时请求 `pthread_cond_wait()`（或 `pthread_cond_timedwait()`）的竞争条件。`mutex` 互斥锁必须是普通锁（`PTHREAD_MUTEX_TIMED_NP`）或者适应锁（`PTHREAD_MUTEX_ADAPTIVE_NP`），且在调用 `pthread_cond_wait()` 前必须由本线程加锁（`pthread_mutex_lock()`），而在更新条件等待队列以前，`mutex` 需保持锁定状态，并在线程挂起进入等待前解锁。在条件满足从而离开 `pthread_`

`cond_wait()` 之前, `mutex` 将被重新加锁, 以与进入 `pthread_cond_wait()` 前的加锁动作对应。
`pthread_cond_wait()` 和 `pthread_cond_timedwait()` 的函数原型分别是:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const
                           struct timespec *abstime)
```

(4) 激发: 激发条件有两种形式: `pthread_cond_signal()` 激活一个等待该条件的线程, 存在多个等待线程时按入队顺序激活其中一个; 而 `pthread_cond_broadcast()` 则激活所有等待线程。

`pthread_cond_signal` 函数的作用是发送一个信号给另外一个正在处于阻塞等待状态的线程, 使其脱离阻塞状态, 继续执行。如果没有线程处在阻塞等待状态, `pthread_cond_signal` 也会成功返回。

使用 `pthread_cond_signal` 不会有“惊群现象”(每当有资源可用, 所有的进程/线程都来竞争资源)产生, 它最多只给一个线程发信号。假如有多个线程正在阻塞等待着这个条件变量的话, 那么根据各等待线程优先级的高低确定哪个线程会接收到信号并开始继续执行; 如果各线程优先级相同, 则根据等待时间的长短来确定哪个线程获得信号。但无论如何, 一个 `pthread_cond_signal` 调用最多发一次信号。

【例 9.12】条件变量的初次使用。

```
#include <iostream>
#include <pthread.h>
using namespace std;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER; /* 初始构造条件变量 */
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER; /* 初始构造锁 */
int x = 10;
int y = 20;
void *func1(void *arg){
    cout<<"func1 start"<<endl;
    pthread_mutex_lock(&qlock);
    while(x<y)
    {
        pthread_cond_wait(&qready,&qlock);
    }
    pthread_mutex_unlock(&qlock);
    sleep(3);
    cout<<"func1 end"<<endl;
}
void *func2(void *arg){
    cout<<"func2 start"<<endl;
    pthread_mutex_lock(&qlock);
    x = 20;
    y = 10;
    cout<<"has change x and y"<<endl;
    pthread_mutex_unlock(&qlock);
    if(x > y){
        pthread_cond_signal(&qready);
    }
}
```

```

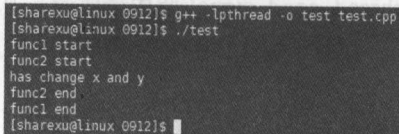
    }
    cout<<"func2 end"<<endl;
}

int main(int argc,char **argv){
    pthread_t tid1,tid2;
    int iRet;
    iRet = pthread_create(&tid1,NULL,func1,NULL);
    if(iRet){
        cout<<"pthread 1 create error"<<endl;
        return iRet;
    }
    sleep(2);
    iRet = pthread_create(&tid2,NULL,func2,NULL);
    if(iRet){
        cout<<"pthread 2 create error"<<endl;
        return iRet;
    }
    sleep(5);
    return 0;
}

```

程序的执行结果如图 9-10 所示。

例 9.12 中创建了 2 个线程，分别执行 func1 函数和 func2 函数。为了确保线程 1 先执行，这里在创建线程 2 之前，先 sleep 2 秒，以便可以更清晰地看到条件变量的作用。



```

[sharex@linux 0912]$ g++ -lpthread -o test test.cpp
[sharex@linux 0912]$ ./test
func1 start
func2 start
has change x and y
func2 end
func1 end
[sharex@linux 0912]$

```

图 9-10 例 9.12 中程序的执行结果

```

    iRet = pthread_create(&tid1,NULL,func1,NULL);
    if(iRet){
        cout<<"pthread 1 create error"<<endl;
        return iRet;
    }
    sleep(2);
    iRet = pthread_create(&tid2,NULL,func2,NULL);
    if(iRet){
        cout<<"pthread 2 create error"<<endl;
        return iRet;
    }
}

```

在 func1 函数中，如果全局变量 $x < y$ ，则阻塞等待，否则继续。由于线程 1 先执行，而且 x 初始化的值是 10， y 初始化的值是 20，所以这里会阻塞等待。

```

int x = 10;
int y = 20;
void *func1(void *arg){
    cout<<"func1 start"<<endl;
    pthread_mutex_lock(&qlock);
    while(x<y)

```



```

{
    pthread_cond_wait(&qready,&qlock);
}
pthread_mutex_unlock(&qlock);
sleep(3);
cout<<"func1 end"<<endl;
}

```

在 func2 函数中, 把 x 和 y 的值都改变了, 导致 $x > y$, 这时, 再用 pthread_cond_signal 函数发送一个信号给另外一个正在处于阻塞等待状态的线程, 使其脱离阻塞状态, 继续执行。这里线程 1 就会收到信号, 并且继续执行。

```

void *func2(void *arg){
    cout<<"func2 start"<<endl;
    pthread_mutex_lock(&qlock);
    x = 20;
    y = 10;
    cout<<"has change x and y"<<endl;
    pthread_mutex_unlock(&qlock);
    if(x > y){
        pthread_cond_signal(&qready);
    }
    cout<<"func2 end"<<endl;
}

```

因此, 程序的执行结果是 func1 先开始, 然后 func2 执行; 当 func2 执行完毕后, 继续执行 func1。

条件变量特别适用于多个线程等待某个条件的发生。如果不使用条件变量, 那么每个线程就需要不断尝试获得互斥锁并检查条件是否发生, 这样大大浪费了系统的资源。

可见, 条件变量是个好东西, 不过如果用得不好, 也容易产生问题。例 9.13 举了出租车的例子。

【例 9.13】出租车的疑问。

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <iostream>
#include <pthread.h>
using namespace std;
/* 提示出租车到达的条件变量 */
pthread_cond_t taxiCond = PTHREAD_COND_INITIALIZER;
/* 同步锁 */
pthread_mutex_t taxiMutex = PTHREAD_MUTEX_INITIALIZER;

```

出租车到达的函数, 就是来了之后就通知乘客。

```

void * traveler_arrive(void * name){
    cout<<"Traveler: "<<(char *)name<<" needs a taxi now!"<<endl;
    pthread_mutex_lock(&taxiMutex);
    pthread_cond_wait(&taxiCond,&taxiMutex);
    pthread_mutex_unlock(&taxiMutex);
    cout<<"Traveler: "<<(char *)name<<" now got a taxi!"<<endl;
    pthread_exit((void*)0);
}

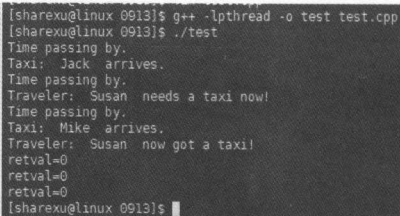
void * taxi_arrive(void * name){
    cout<<"Taxi: "<<(char *)name<<" arrives."<<endl;
    pthread_cond_signal(&taxiCond);
    pthread_exit((void*)0);
}

int main(){
    pthread_t tids[3];
    int iRet = pthread_create(&tids[0],NULL,taxi_arrive,(void*)(" Jack "));
    if(iRet){
        printf("pthread_create error: iRet=%d\n",iRet);
        return iRet;
    }
    printf("Time passing by.\n");
    sleep(1);
    iRet = pthread_create(&tids[1],NULL,traveler_arrive,(void*)(" Susan "));
    if(iRet){
        printf("pthread_create error: iRet=%d\n",iRet);
        return iRet;
    }
    printf("Time passing by.\n");
    sleep(1);
    iRet = pthread_create(&tids[2],NULL,taxi_arrive,(void*)(" Mike "));
    if(iRet){
        printf("pthread_create error: iRet=%d\n",iRet);
        return iRet;
    }
    printf("Time passing by.\n");
    sleep(1);
    void *retval;
    for(int i=0;i<3;i++){
        iRet=pthread_join(tids[i],&retval);
        if (iRet){
            printf("pthread_join error: iRet=%d\n",iRet);
            return iRet;
        }
        printf("retval=%ld\n",(long)retval);
    }
    return 0;
}

```

程序的执行结果如图 9-11 所示。

例 9.13 中模拟了出租车与乘客的到达情况。程序中一共创建了 3 个线程，2 个是出租车（调用出租车到达的函数），1 个是乘客（调用乘客到达的函数），顺序分别是出租车 Jack 先到，过了 1 秒，乘客 Susan 再到，再过 1 秒，出租车 Mike 最后到。



```
[sharexu@linux 0913]$ g++ -lpthread -o test test.cpp
[sharexu@linux 0913]$ ./test
Time passing by.
Taxi: Jack arrives.
Time passing by.
Traveler: Susan needs a taxi now!
Time passing by.
Taxi: Mike arrives.
Traveler: Susan now got a taxi!
retval=0
retval=0
retval=0
[sharexu@linux 0913]$
```

图 9-11 例 9.13 中程序的执行结果

```
pthread_t tids[3];
int iRet = pthread_create(&tids[0],NULL,taxi_arrive,(void*)(" Jack "));
if(iRet){
    printf("pthread_create error: iRet=%d\n",iRet);
    return iRet;
}
printf("Time passing by.\n");
sleep(1);
iRet = pthread_create(&tids[1],NULL,traveler_arrive,(void*)(" Susan "));
if(iRet){
    printf("pthread_create error: iRet=%d\n",iRet);
    return iRet;
}
printf("Time passing by.\n");
sleep(1);
iRet = pthread_create(&tids[2],NULL,taxi_arrive,(void*)(" Mike "));
if(iRet){
    printf("pthread_create error: iRet=%d\n",iRet);
    return iRet;
}
printf("Time passing by.\n");
sleep(1);
```

程序有一个条件变量，用于提示出租车到达，还有一个同步锁，代码如下：

```
/* 提示出租车到达的条件变量 */
pthread_cond_t taxiCond = PTHREAD_COND_INITIALIZER;
/* 同步锁 */
pthread_mutex_t taxiMutex = PTHREAD_MUTEX_INITIALIZER;
```

乘客到达的函数，就是来了之后就等车。

```
void * traveler_arrive(void * name){
    cout<<"Traveler: "<<(char *)name<<" needs a taxi now!"<<endl;
    pthread_mutex_lock(&taxiMutex);
    pthread_cond_wait(&taxiCond,&taxiMutex);
    pthread_mutex_unlock(&taxiMutex);
    cout<<"Traveler: "<<(char *)name<<" now got a taxi!"<<endl;
    pthread_exit((void*)0);
}
```

出租车到达的函数，就是来了之后就通知乘客。

```

void * taxi_arrive(void * name){
    cout<<"Taxi: "<<(char *)name<<" arrives."<<endl;
    pthread_cond_signal(&taxiCond);
    pthread_exit((void*)0);
}

```

那程序的执行结果显示：Jack 到了站台一看没人，触发的条件变量被直接复位，于是 Jack 排在等待队列里面。来迟 1 秒的 Susan 到了站台却看不到在那里等待的 Jack，只能等待，直到 Mike 开车赶到，重新触发条件变量，Susan 才上了车。疑问有两个，Susan 来的时候明明有 Jack 的出租车在，两边却都在白等；Susan 最后是上了 Jack 的车，还是上了 Mike 的车？

这里其实是没有掌握好触发的时机。我们可以增加一个计数器记录等待线程的个数，在决定触发条件变量前检查该变量即可。改后程序如例 9.14 所示。

【9.14】完美的出租车。

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include <iostream>
#include <pthread.h>
using namespace std;

/* 提示出租车到达的条件变量 */
pthread_cond_t taxiCond = PTHREAD_COND_INITIALIZER;
/* 同步锁 */
pthread_mutex_t taxiMutex = PTHREAD_MUTEX_INITIALIZER;

int travelerCount=0;

void * traveler_arrive(void * name){
    cout<<"Traveler: "<<(char *)name<<" needs a taxi now!"<<endl;
    pthread_mutex_lock(&taxiMutex);
    travelerCount++;
    pthread_cond_wait(&taxiCond,&taxiMutex);
    pthread_mutex_unlock(&taxiMutex);
    cout<<"Traveler: "<<(char *)name<<" now got a taxi!"<<endl;
    pthread_exit((void*)0);
}

void * taxi_arrive(void * name){
    cout<<"Taxi: "<<(char *)name<<" arrives."<<endl;
    while(1){
        pthread_mutex_lock(&taxiMutex);
        if(travelerCount>0){
            pthread_cond_signal(&taxiCond);

```

```

        pthread_mutex_unlock(&taxiMutex);
        break;
    }
    pthread_mutex_unlock(&taxiMutex);
}
pthread_exit((void*)0);
}

int main(){
    pthread_t tids[3];
    int iRet = pthread_create(&tids[0],NULL,taxi_arrive,(void*)(" Jack "));
    if(iRet){
        printf("pthread_create error: iRet=%d\n",iRet);
        return iRet;
    }
    printf("Time passing by.\n");
    sleep(1);
    iRet = pthread_create(&tids[1],NULL,traveler_arrive,(void*)(" Susan "));
    if(iRet){
        printf("pthread_create error: iRet=%d\n",iRet);
        return iRet;
    }
    printf("Time passing by.\n");
    sleep(1);
    iRet = pthread_create(&tids[2],NULL,taxi_arrive,(void*)(" Mike "));
    if(iRet){
        printf("pthread_create error: iRet=%d\n",iRet);
        return iRet;
    }
    printf("Time passing by.\n");
    sleep(1);

    void *retval;
    for(int i=0;i<3;i++){
        iRet=pthread_join(tids[i],&retval);
        if (iRet){
            printf("pthread_join error: iRet=%d\n",iRet);
            return iRet;
        }
        printf("retval=%ld\n",(long)retval);
    }
    return 0;
}

```

程序的执行结果如图 9-12 所示。

这样, Susan 一来就发现了 Jack 的车, 双方都不用白等。

对比例 9.13, 程序有以下不同。

```

[sharexu@linux 0914]$ g++ -lpthread -o test test.cpp
[sharexu@linux 0914]$ ./test
Time passing by.
Taxi: Jack arrives.
Time passing by.
Traveler: Susan needs a taxi now!
Traveler: Susan now got a taxi!
Time passing by.
Taxi: Mike arrives.
retval=0
retval=0
retval=0
[sharexu@linux 0914]$

```

图 9-12 例 9.14 中程序的执行结果

增加了一个记录旅客数量的变量。旅客到达时，这个变量会加 1，代码如下：

```
int travelerCount=0;
void * traveler_arrive(void * name){
    cout<<"Traveler: "<<(char *)name<<" needs a taxi now!"<<endl;
    pthread_mutex_lock(&taxiMutex);
    travelerCount++;
    pthread_cond_wait(&taxiCond,&taxiMutex);
    pthread_mutex_unlock(&taxiMutex);
    cout<<"Traveler: "<<(char *)name<<" now got a taxi!"<<endl;
    pthread_exit((void*)0);
}
```

出租车到达的函数中，加了个 while 永真循环，这样可以保证先来的 Jack 也能检测到是否有顾客到达。在循环中判断顾客人数是否大于 0，如果大于 0，则通知 Jack。这样，先来的 Jack 便接到了顾客，代码如下：

```
void * taxi_arrive(void * name){
    cout<<"Taxi: "<<(char *)name<<" arrives."<<endl;
    while(1){
        pthread_mutex_lock(&taxiMutex);
        if(travelerCount>0){
            pthread_cond_signal(&taxiCond);
            pthread_mutex_unlock(&taxiMutex);
            break;
        }
        pthread_mutex_unlock(&taxiMutex);
    }
    pthread_exit((void*)0);
}
```

4. 读写锁

在一些程序中存在读者写者问题，也就是说，对某些资源的访问会存在两种可能的情况，一种是访问必须是排他性的，就是独占的意思，这称作写操作；另一种情况就是访问方式可以是共享的，就是说可以有多个线程同时去访问某个资源，这种就称作读操作。这个问题模型是从对文件的读写操作中引申出来的。

(1) 读写锁比起互斥锁具有更高的适用性与并行性，可以有多个线程同时占用读模式的读写锁，但是只能有一个线程占用写模式的读写锁，读写锁的 3 种状态如下所述。

1) 当读写锁是写加锁状态时，在这个锁被解锁之前，所有试图对这个锁加锁的线程都会被阻塞。

2) 当读写锁在读加锁状态时，所有试图以读模式对它进行加锁的线程都可以得到访问权，但是以写模式对它进行加锁的线程将会被阻塞。

3) 当读写锁在读模式的锁状态时，如果有另外的线程试图以写模式加锁，读写锁通常

会阻塞随后的读模式锁的请求，这样可以避免读模式锁长期占用，而等待的写模式锁请求则长期阻塞。

读写锁最适用于对数据结构的读操作次数多于写操作次数的场合，因为读模式锁定时可以共享，而写模式锁定时只能由某个线程独占资源，因而读写锁也可以叫作共享-独占锁。

处理读者-写者问题的两种常见策略是强读者同步（strong reader synchronization）和强写者同步（strong writer synchronization）。在强读者同步中，总是给读者更高的优先权，只要写者当前没有进行写操作，读者就可以获得访问权限；而在强写者同步中，则往往将优先权交付给写者，而读者只能等到所有正在等待的或者正在执行的写者结束以后才能执行。关于读者-写者模型，由于读者往往会要求查看最新的信息记录，例如航班订票系统往往会使用强写者同步策略，而图书馆查阅系统则采用强读者同步策略。

（2）读写锁机制是由 POSIX 提供的，如果写者没有持有读写锁，那么所有的读者都可以持有这把锁，而一旦有某个写者阻塞在上锁的时候，那么就由 POSIX 系统来决定是否允许读者获取该锁。

接下来我们来看读写锁相关的函数使用，如下所述。

1) 初始化和销毁读写锁。

对于读写锁变量的初始化可以有两种方式，一种是通过给一个静态分配的读写锁赋予常值 `PTHREAD_RWLOCK_INITIALIZER` 来初始化它；另一种方法就是通过调用 `pthread_rwlock_init()` 来动态地初始化。而当某个线程不再需要读写锁的时候，可以通过调用 `pthread_rwlock_destroy` 来销毁该锁。函数原型如下：

```
int pthread_rwlock_init(pthread_rwlock_t *rwptr, const pthread_rwlockattr_t
*attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwptr);
```

这两个函数如果执行成功均返回 0，如果出错则返回错误码。

在释放某个读写锁占用的内存之前，要先通过 `pthread_rwlock_destroy` 对读写锁进行清理，释放由 `pthread_rwlock_init` 所分配的资源。

在初始化某个读写锁的时候，如果属性指针 `attr` 是个空指针的话，表示使用默认属性；如果想要使用非默认属性，则要使用到下面的两个函数：

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

同样的，这两个函数如果执行成功则返回 0，失败则返回错误码。

这里还需要说明的是，当初始化读写锁完毕以后，该锁就处于一种非锁定状态。

数据类型为 `pthread_rwlockattr_t` 的某个属性对象一旦初始化了，就可以通过不同的函数调用来启用或禁用某个特定的属性。

2) 获取和释放读写锁。

读写锁的数据类型是 `pthread_rwlock_t`，如果这个数据类型中的某个变量是静态分配的，那么可以通过给它赋予常值 `PTHREAD_RWLOCK_INITIALIZER` 来初始化它。`pthread_rwlock_rdlock()` 用来获取读出锁，如果相应的读出锁已经被某个写入者占有，那么就阻塞调用线程。

`pthread_rwlock_wrlock()` 以获取一个写入锁，如果相应的写入锁已经被其他写入者或者读出者占有（一个或多个），那么就阻塞该调用线程；`pthread_rwlock_unlock()` 用来释放一个读出或者写入锁。函数原型如下：

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwptr);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwptr);
int pthread_rwlock_unlock(pthread_rwlock_t *rwptr);
```

这 3 个函数若调用成功则返回 0，失败则返回错误码。要注意的是，其中获取锁的两个函数的操作都是阻塞操作，也就是说获取不到锁的话，那么调用线程不是立即返回，而是阻塞执行。有写情况下，这种阻塞式的获取锁的方式可能不是很适用，所以，接下来引入两个采用非阻塞方式获取读写锁的函数 `pthread_rwlock_tryrdlock()` 和 `pthread_rwlock_trywrlock()`，非阻塞方式下获取锁的时候，如果不能马上获取到，就会立即返回一个 `EBUSY` 错误提示，而不是把调用线程投入到睡眠等待。函数原型如下：

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwptr);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwptr);
```

同样，这两个函数调用成功则返回 0，失败则返回错误码。

下面用个实例（见例 9.15）来展示下读写锁的使用。

【例 9.15】读写锁的使用。

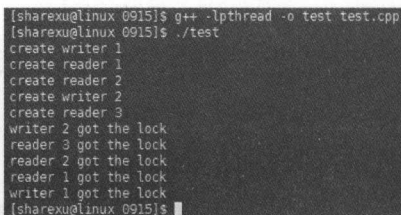
```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#define THREADNUM 5
pthread_rwlock_t rwlock;
void *readers(void *arg){
    pthread_rwlock_rdlock(&rwlock);
    printf("reader %ld got the lock\n", (long)arg);
    pthread_rwlock_unlock(&rwlock);
    pthread_exit((void*)0);
}
void *writers(void *arg){
    pthread_rwlock_wrlock(&rwlock);
    printf("writer %ld got the lock\n", (long)arg);
    pthread_rwlock_unlock(&rwlock);
    pthread_exit((void*)0);
}
int main(int argc, char **argv){
```

```

int iRet, i;
pthread_t writer_id, reader_id;
pthread_attr_t attr;
int nreadercount = 1, nwritercount = 1;
iRet = pthread_rwlock_init(&rwlock, NULL);
if (iRet) {
    fprintf(stderr, "init lock failed\n");
    return iRet;
}
pthread_attr_init(&attr);
/*pthread_attr_setdetachstate 用来设置线程的分离状态, 也就是说一个线程怎么样终止自己,
  状态设置为 PTHREAD_CREATE_DETACHED, 表示以分离状态启动线程 */
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
for (i = 0; i < THREADNUM; i++){
    if (i % 3) {
        pthread_create(&reader_id, &attr, readers, (void *)nreadercount);
        printf("create reader %d\n", nreadercount++);
    } else {
        pthread_create(&writer_id, &attr, writers, (void *)nwritercount);
        printf("create writer %d\n", nwritercount++);
    }
}
sleep(5); /*sleep 是为了等待另外的线程的执行 */
return 0;
}

```

程序的执行结果如图 9-13 所示。



```

[sharex@linux 0915]$ g++ -lpthread -o test test.cpp
[sharex@linux 0915]$ ./test
create writer 1
create reader 1
create reader 2
create writer 2
create reader 3
writer 2 got the lock
reader 3 got the lock
reader 2 got the lock
reader 1 got the lock
writer 1 got the lock
[sharex@linux 0915]$

```

图 9-13 例 9.15 中程序的执行结果

在例 9.15 中, 定义了一个全局的读写锁。在 main 函数中, 初始化了读写锁, 创建了 5 个线程, 其中有 3 个调用了 readers 函数, 有 2 个调用了 writers 函数。而 readers 函数中, 加上读锁, 输出提示语后可解锁; 而 writers 函数中, 加上写锁, 输出提示语后解锁。读加锁的线程比写加锁的线程多, 方便看到哪个线程容易获得锁。执行结果展示, 当读写锁是写加锁状态时, 在这个锁被解锁之前, 所有试图对这个锁加锁的线程都会被阻塞; 当读写锁在读加锁状态时, 所有试图以读模式对它进行加锁的线程都可以得到访问权, 但是以写模式对它进行加锁的线程将会被阻塞。

5. 信号量

线程还可以通过信号量来实现通信。信号量和互斥锁的区别: 互斥锁只允许一个线程进入临界区, 而信号量允许多个线程同时进入临界区。要使用信号量同步, 需要包含头文件 semaphore.h。信号量函数的名字都以 "sem_" 打头。线程使用的基本信号量函数有以下 4 个。

(1) sem_init 函数。

该函数用于创建信号量, 其原型如下:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

该函数用于初始化由 `sem` 指向的信号对象，设置它的共享选项，并给它一个初始的整数值。`pshared` 控制信号量的类型，如果其值为 0，就表示这个信号量是当前进程的局部信号量，否则信号量就可以在多个进程之间共享。`value` 为 `sem` 的初始值。调用成功时返回 0，失败返回 -1。

(2) `sem_wait` 函数。

该函数用于以原子操作的方式将信号量的值减 1。原子操作就是，如果两个线程企图同时给一个信号量加 1 或减 1，它们之间不会互相干扰，函数的原型如下：

```
int sem_wait(sem_t *sem);
```

`sem` 指向的对象是由 `sem_init` 调用初始化的信号量。调用成功时返回 0，失败返回 -1。

(3) `sem_post` 函数。

该函数用于以原子操作的方式将信号量的值加 1，函数的原型如下：

```
int sem_post(sem_t *sem);
```

与 `sem_wait` 一样，`sem` 指向的对象是由 `sem_init` 调用初始化的信号量。调用成功时返回 0，失败返回 -1。

(4) `sem_destroy` 函数。

该函数用于对用完的信号量进行清理，函数的原型如下：

```
int sem_destroy(sem_t *sem);
```

成功时返回 0，失败时返回 -1。

下面用例 9.16 演示如何用信号量同步，模拟一个窗口服务系统。

【例 9.16】用信号量模拟窗口服务系统。

```
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define CUSTOMER_NUM 10
/* @Scene: 某行业营业厅同时只能服务两个顾客。
 * 有多个顾客到来，顾客如果发现服务窗口已满，就等待，
 * 如果有可用的服务窗口，就接受服务。 */
/* 将信号量定义为全局变量，方便多个线程共享 */
sem_t sem;
/* 每个线程要运行的例程 */
void * get_service(void *thread_id){
/* 注意：立即保存 thread_id 的值，因为 thread_id 是对主线程中循环变量 i 的引用，它可能马上被修改 */
    int customer_id = *((int *)thread_id);
```

```

    if(sem_wait(&sem) == 0) {
        usleep(100); /* service time: 100ms */
        printf("customer %d receive service ...\n", customer_id);
        sem_post(&sem);
    }
}

int main(int argc, char *argv[]){
    /* 初始化信号量, 初始值为 2, 表示有两个顾客可以同时接受服务 */
    sem_init(&sem, 0, 2);
    /* 为每个顾客定义一个线程 id */
    pthread_t customers[CUSTOMER_NUM];
    int i, iRet;
    /* 为每个顾客生成一个线程 */
    for(i = 0; i < CUSTOMER_NUM; i++){
        int customer_id = i;
        iRet = pthread_create(&customers[i], NULL, get_service, &customer_id);
        if(iRet){
            perror("pthread_create");
            return iRet;
        }
        else{
            printf("Customer %d arrived.\n", i);
        }
        usleep(10);
    }
    /* 等待所有顾客的线程结束 */
    /* 注意: 这地方不能再用 i 做循环变量, 因为可能线程正在访问 i 的值 */
    int j;
    for(j = 0; j < CUSTOMER_NUM; j++) {
        pthread_join(customers[j], NULL);
    }
    /* 销毁信号量 */
    sem_destroy(&sem);
    return 0;
}

```

程序的执行结果如图 9-14 所示。

例 9.16 中模拟的是一个营业厅只能同时服务两个顾客的情况, 当有多个顾客到来时, 每个顾客如果发现服务窗口已满, 就等待, 当有可用的服务窗口时, 就接受服务。

将信号量定义为全局变量, 方便多个线程共享。

```
sem_t sem;
```

main 函数中, 初始化信号量, 初始值为 2, 表示有两个顾客可以同时接受服务, 代码如下:

```

[sharexu@linux 0916]$ g++ -lpthread -o test test.cpp
[sharexu@linux 0916]$ ./test
Customer 0 arrived.
Customer 1 arrived.
customer 0 receive service ...
Customer 2 arrived.
Customer 3 arrived.
customer 1 receive service ...
Customer 4 arrived.
Customer 5 arrived.
customer 3 receive service ...
customer 3 receive service ...
Customer 6 arrived.
Customer 7 arrived.
customer 4 receive service ...
customer 5 receive service ...
Customer 8 arrived.
Customer 9 arrived.
customer 6 receive service ...
customer 7 receive service ...
customer 8 receive service ...
customer 9 receive service ...
[sharexu@linux 0916]$

```

图 9-14 例 9.16 中程序的执行结果


```
sem_init(&sem, 0, 2);
```

为每个顾客生成一个线程，好像顾客陆续到来一样，并且每个线程都调用 `get_service` 函数。注意，`get_service` 函数要立即保存 `thread_id` 的值，因为 `thread_id` 是对主线程中循环变量 `i` 的引用，它可能马上被修改。`if (sem_wait (&sem) == 0)` 表示当前信号量大于 0，可以为该顾客服务，并将信号量 -1，服务完成后，就得调用 `sem_post` 把信号量加 1，以便继续为其他顾客服务，代码如下：

```
void * get_service(void *thread_id){
    int customer_id = *((int *)thread_id);
    if(sem_wait(&sem) == 0) {
        usleep(100); /* service time: 100ms */
        printf("customer %d receive service ...\n", customer_id);
        sem_post (&sem);
    }
}
```

多线程的几种同步方式已介绍完毕，大家应该结合实际情况，选择适当的方式进行使用。

9.5 多线程重入

前面介绍了各种同步方式，其实都是为了解决“函数不可重入”的问题。所谓“可重入函数”，是指可以由多于一个任务并发使用，而不必担心数据错误的函数。相反，“不可重入函数”则是只能由一个任务所占用，除非能确保函数的互斥（或者使用信号量，或者在代码的关键部分禁用中断）。可重入函数可以在任意时刻被中断，稍后再继续运行，且不会丢失数据。可重入函数要在使用本地变量或使用全局变量时保护自己的数据。

(1) 可重入函数有以下特点。

- 1) 不为连续的调用持有静态数据。
- 2) 不返回指向静态数据的指针。
- 3) 所有数据都由函数的调用者提供。
- 4) 使用本地数据，或者通过制作全局数据的本地副本来保护全局数据。
- 5) 如果必须访问全局变量，要利用互斥锁、信号量等来保护全局变量。
- 6) 绝不调用任何不可重入函数。

(2) 不可重入函数有以下特点。

- 1) 函数中使用了静态变量，无论是全局静态变量还是局部静态变量。
- 2) 函数返回静态变量。
- 3) 函数中调用了不可重入函数。
- 4) 函数体内使用了静态的数据结构。

5) 函数体内调用了 `malloc()` 或者 `free()` 函数。

6) 函数体内调用了其他标准 I/O 函数。

在一个多线程程序里, 默认情况下, 只有一个 `errno` 变量供所有的线程共享。在一个线程准备获取刚才的错误代码时, 该变量很容易被另一个线程中的函数调用所改变。类似的问题还存在于 `fputs` 之类的函数中, 这些函数通常用一个单独的全局性区域来缓存输出数据。

为解决这个问题, 需要使用可重入的例程。可重入代码可以被多次调用而仍然正常工作。编写的多线程程序, 通过定义宏 `_REENTRANT` 来告诉编译器需要可重入功能, 这个宏的定义必须出现于程序中的任何 `#include` 语句之前。

(3) `_REENTRANT` 为我们做三件事情, 并且做得非常优雅:

1) 它会对部分函数重新定义它们的可安全重入的版本, 这些函数名字一般不会发生改变, 只是会在函数名后面添加 `_r` 字符串, 如函数名 `gethostbyname` 变成 `gethostbyname_r`。

2) `stdio.h` 中原来以宏的形式实现的一些函数将变成可安全重入函数。

3) 在 `error.h` 中定义的变量 `error` 现在将成为一个函数调用, 它能够以一种安全的多线程方式来获取真正的 `errno` 的值。

9.6 本章小结

本章主要介绍了多线程的使用, 包括如何利用各种同步方式, 使得线程安全、高效等。学会了多线程, 就可以提高应用程序响应, 使多 CPU 系统更加有效并改善程序结构。

前面提到的进程, 相关用法将在第 10 章详细地展开。

进 程

前面各章都或多或少地提到了进程，究竟进程是什么？进程，是计算机中处于运行中程序的实体。以前，进程是最小的运行单位；有了线程之后，线程成为最小的运行单位，而进程则是线程的容器。程序本身只是指令、数据及其组织形式的描述，进程才是程序（指令和数据）的真正运行实例。多个进程可与同一个程序相关联，而每个进程则是以同步或异步的方式独立运行的。本章主要探讨进程的相关内容。

10.1 程序与进程

先来看下 Linux 的进程结构，进程结构一般由 3 部分组成：代码段、数据段和堆栈段。代码段是用于存放程序代码的数据，假如机器中有数个进程运行相同的一个程序，那么它们就可以使用同一个代码段。而数据段则存放程序的全局变量、常量和静态变量。堆栈段中的栈用于函数调用，它存放着函数的参数、函数内部定义的局部变量。堆栈段还包括了进程控制块（Process Control Block, PCB）。PCB 处于进程核心堆栈的底部，不需要额外分配空间。PCB 是进程存在的唯一标识，系统通过 PCB 的存在而感知进程的存在。系统通过 PCB 对进程进行管理和调度。PCB 包括创建进程、执行程序、退出进程以及改变进程的优先级等。

既然进程是一个程序的执行过程，那么程序又是怎么转化为进程的呢？

一般情况下 Linux 下 C++ 程序的生成可分为 4 个阶段：预编译、编译、汇编、链接。编译器 g++ 经过预编译、编译、汇编 3 个步骤将源程序文件转换为目标文件（第 4 章有如何生成目标文件的详细过程，有需要者可随时翻阅）。如果程序有多个目标文件或者程序

使用了库函数，编译器还要将所有的目标文件或所需要的库链接起来，最后形成可执行程序。当程序执行时，操作系统将可执行程序复制到内存中。一般程序转换为进程分以下几个步骤：

- ①内核将程序读入内存，为程序分配内存空间；
- ②内核为该进程分配进程标识符（PID）和其他所需资源；
- ③内核为进程保存 PID 及相应的状态信息，把进程放到运行队列中等待执行，程序转化为进程后就可以被操作系统的调度程序调度执行了。

每个进程在系统中都有唯一的一个 ID 标识它，这个 ID 就是进程标识符（PID）。因为其唯一性，所以系统可以根据它准确定位到一个进程。进程标识符的类型为 `pid_t`，其本质上是一个无符号整型的类型别名。所谓程序，不过是指可运行的二进制代码文件，把这种文件加载到内存中运行就得到了一个进程。同一个程序文件可以被加载多次成为不同的进程。因此，进程与进程标识符之间是一一对应的关系，而与程序文件之间是多对一的关系，如图 10-1 所示。

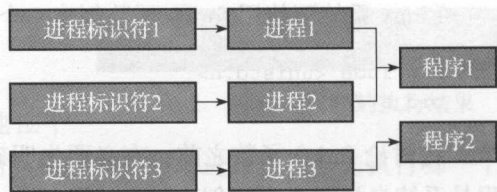


图 10-1 进程标识符与进程、程序之间的关系

10.2 进程的创建与结束

进程的创建有两种方式：一种是由操作系统创建，一种是由父进程创建。

在系统启动时，操作系统会创建一些进程，它们承担着管理和分配系统资源的任务，这些进程通常被称为系统进程。系统允许一个进程创建新进程（即为子进程），子进程还可以创建新的子进程，形成进程树结构。整个 Linux 系统的所有进程也是一个树形结构。树根是系统自动构造的，即在内核态下执行的 0 号进程，它是所有进程的祖先。由 0 号进程创建 1 号进程（内核态），1 号负责执行内核的部分初始化工作及进行系统配置，并创建若干个用于高速缓存和虚拟内存管理的内核线程。随后，1 号进程调用 `execve()` 运行可执行程序 `init`，并演变成用户态 1 号进程，即 `init` 进程。它按照配置文件 `/etc/inittab` 的要求，完成系统启动工作，创建编号为 1 号、2 号……的若干终端注册进程 `getty`。每个 `getty` 进程设置其进程组标识号，并检测配置到系统终端的接口线路。当检测到来自终端的连接信号时，`getty` 进程将通过函数 `execve()` 执行注册程序 `login`，此时用户就可输入注册名和密码进入登录过程，如果成功，由 `login` 程序再通过函数 `execv()` 执行 `shell`，该 `shell` 进程接收 `getty` 进程的 `pid`，取代原来的 `getty` 进程。再由 `shell` 直接或间接地产生其他进程。

上述过程可描述为：0 号进程→1 号内核进程→1 号内核线程→1 号用户进程（`init` 进程）→`getty` 进程→`shell` 进程。

注意，上述过程描述中提到：1 号内核进程调用执行 `init` 并演变成 1 号用户态进程（`init`

进程), 这里前者是 `init` 是函数, 后者是进程。两者容易混淆, 区别如下所述。

- (1) `init()` 函数在内核态运行, 是内核代码。
- (2) `init` 进程是内核启动并运行的第一个用户进程, 运行在用户态下。
- (3) `init()` 函数调用 `execve()` 从文件 `/etc/inittab` 中加载可执行程序 `init` 并执行, 这个过程并没有使用调用 `do_fork()`, 因此两个进程都是 1 号进程。

1. 进程的创建——`fork()` 函数

Linux 系统允许任何一个用户进程创建一个子进程, 创建成功后, 子进程将存在于系统之中, 并且独立于父进程。该子进程可以接受系统调度, 可以得到分配的系统资源。系统也可以检测到子进程的存在, 并且赋予它与父进程同样的权利。

Linux 系统下使用 `fork()` 函数创建一个子进程, 其函数原型如下:

```
#include <unistd.h>
pid_t fork(void);
```

在讨论 `fork()` 函数之前, 有必要先明确父进程和子进程两个概念。除了 0 号进程 (该进程是系统自举时由系统创建的) 以外, Linux 系统中的任何一个进程都是由其他进程创建的。创建新进程的进程, 即调用 `fork()` 函数的进程就是父进程, 而新创建的进程就是子进程。

`fork()` 函数不需要参数, 返回值是一个进程标识符 (PID)。对于返回值, 有以下 3 种情况。

- (1) 对于父进程, `fork()` 函数返回新创建的子进程的 ID。
- (2) 对于子进程, `fork()` 函数返回 0。
- (3) 如果创建出错, 则 `fork()` 函数返回 -1。

`fork()` 函数会创建一个新的进程, 并从内核中为此进程分配一个新的可用的进程标识符 (PID), 之后, 为这个新进程分配进程空间, 并将父进程的进程空间中的内容复制到子进程的进程空间中, 包括父进程的数据段和堆栈段, 并且和父进程共享代码段。这时候, 系统中又多了一个进程, 这个进程和父进程一模一样, 两个进程都要接受系统的调度。由于在复制时复制了父进程的堆栈段, 所以两个进程都停留在了 `fork()` 函数中, 等待返回。因此, `fork()` 函数会返回两次, 一次是在父进程中返回, 另一次是在子进程中返回, 这两次的返回值是不一样的。

下面给出的示例程序例 10.1 用来创建一个子进程, 该程序在父进程和子进程中分别输出不同的内容。

【例 10.1】创建一个子进程。

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int main(void){
    pid_t pid;
```

```

pid = fork();
if (pid < 0){
    perror("fail to fork");
    exit(-1);
}
else if (pid == 0){
    /* 子进程 */
    printf("Sub-process, PID: %u, PPID: %u\n", getpid(), getppid());
}
else{ /* 父进程 */
    printf("Parent, PID: %u, Sub-process PID: %u\n", getpid(), pid);
    sleep(2);
}
return 0;
}

```

```

[sharex@linux 1001]$ ./test
Parent, PID: 24347, Sub-process PID: 24348
Sub-process, PID: 24348, PPID: 24347

```

程序的执行结果如图 10-2 所示。

图 10-2 例 10.1 程序的执行结果

由于创建的新进程和父进程在系统看来是地位平等的两个进程，运行机会也是一样的，故不能够对其执行先后顺序进行假设，先执行哪一个进程取决于系统的调度算法。例 10.1 中为了让父进程不那么快结束，所以加了 `sleep(2)` 语句，休眠 2s 再结束。这样方便读者看到父子进程的关系。`getpid()` 是获得当前进程的 `pid`，而 `getppid()` 则是获得父进程的 `id`。

那么父进程和子进程都共享了哪些资源呢？例 10.2 即解答了这个问题。

【例 10.2】父子进程的共享资源。

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int global = 1; /* 初始化的全局变量，存在 data 段 */
int main(void){
    pid_t pid; /* 存储进程 id */
    int stack = 1; /* 局部变量，存在栈中 */
    int *heap; /* 指向堆变量的指针 */
    heap = (int *)malloc(sizeof(int));
    *heap = 3; /* 设置堆中的值是 3 */
    pid = fork(); /* 创建一个新的进程 */
    if (pid < 0){
        perror("fail to fork");
        exit(-1);
    }
    else if (pid == 0){
        /* 子进程，改变变量的值 */
        global++;
        stack++;
        (*heap)++;
        /* 打印出变量的值 */
        printf("In sub-process, global: %d, stack: %d, heap: %d\n", global,

```



```
stack, *heap);
    exit(0);
}
else{
    /* 父进程 */
    sleep(2);/* 休眠 2 秒钟，确保子进程已执行完毕，再执行父进程 */
    printf("In parent-process, global: %d, stack: %d, heap: %d\n", global,
stack, *heap);
}
return 0;
}
```

程序的执行结果如图 10-3 所示。

例 10.2 中定义了一个全局变量 `global`、一个局部变量 `stack` 和一个指针 `heap`。该指针用来指向一块动态分配的内存区域。之后，该程序创建一个子进程，在子进程中修改 `global`、`stack` 和动态分配的内存中变量的值。然后在父、子进程中分别打印出这些变量的值。由于父、子进程的运行顺序是不确定的，因此可以先让父进程额外休眠 2s，以保证子进程先运行。

由于父进程休眠了 2s，子进程先于父进程运行，因此会先在子进程中修改数据段和堆栈段中的内容。因此不难看出，子进程对这些数据段和堆栈段中内容的修改并不会影响到父进程的进程环境。

事实上，子进程完全复制了父进程的地址空间的内容，包括堆栈段和数据段的内容。但是，子进程并没有复制代码段，而是和父进程共用代码段。这样做是合理的，因为子进程可能执行不同的流程来改变数据段和堆栈段，因此需要分开存储父子进程各自的数据段和堆栈段。但是代码段是只读的，不存在被修改的问题，因此代码段可以让父子进程共享，以节省存储空间，如图 10-4 所示。

父进程的资源大部分被 `fork()` 函数所复制，只有小部分是子进程与父进程不同的。子进程继承的资源情况如表 10-1 所示。

```
[sharexu@linux 1002]$ ./test
In sub-process, global: 2, stack: 2, heap: 4
In parent-process, global: 1, stack: 1, heap: 3
[sharexu@linux 1002]$
```

图 10-3 例 10.2 程序的执行结果

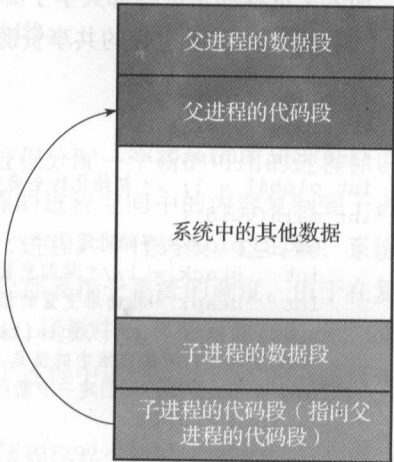


图 10-4 父子进程公用代码段

表 10-1 子进程继承资源的情况

资源	父子进程是否相同	资源	父子进程是否相同
进程 ID	否	实际组 ID	是
实际用户 ID	是	有效组 ID	是
有效用户 ID	是	附加组 ID	是

(续)

资源	父子进程是否相同	资源	父子进程是否相同
进程组 ID	是	.bss 端	是
父进程 ID	否	连接的共享存储段	是
会话 ID	是	存储映射	是
设置用户 ID 标志	是	资源限制	是
设置组 ID 标志	是	tms_utime	否
当前工作目录	是	tms_stime	否
根目录	是	tms_cutime	否
文件权限屏蔽字	是	tms_ustime	否
信号的屏蔽	是	for 函数的返回值	否
打开的文件描述符	是	设置的文件锁	否
数据段	是	未处理的闹钟信号	否
代码段	是	未决信号集	否
堆栈	是		

现在的 Linux 内核在实现 fork() 函数时往往在创建子进程时并不立即复制父进程的数据段和堆栈段,而是当子进程修改这些数据内容时复制操作才会发生,内核才会给予进程分配进程空间,将父进程的内容复制过来,然后继续后面的操作。这样的实现更加合理,对于一些只是为了复制自身完成一些工作的进程来说,这样做的效率会更高。这也是现代操作系统中一个重要的概念——“写时复制”的一个重要体现。

2. 进程的结束——exit() 函数

当一个进程需要退出时,需要调用退出函数。Linux 环境下使用 exit() 函数退出进程,其函数原型如下:

```
#include<stdlib.h>
void exit(int status);
```

exit() 函数的参数表示进程的退出状态,这个状态的值是一个整型,保存在全局变量 \$? 中。

Linux 程序员可以通过 shell 得到已结束进程的结束状态,执行“echo \$?”命令即可。

\$? 是 Linux shell 中的一个内置变量,其中保存的是最近一次运行的进程的返回值。这个返回值有以下 3 种情况:①程序中的 main 函数运行结束,\$? 中保存 main 函数的返回值;②程序运行中调用 exit 函数结束运行,\$? 中保存 exit 函数的参数;③程序异常退出,\$? 中保存异常出错的错误号。

下面的程序用来测试程序的结束状态,该程序接收整型输入,如果输入数字 0 则正常退出。

【例 10.3】从 \$? 获得进程的返回值。

```
#include <iostream>
int main(){
    int i;
    while(1){
        std::cin>>i;
        if (i == 0)
            break;
    }
    return 10;
}
```

程序执行结果如图 10-5 所示。

注意：首先，shell 内置变量 \$? 保存的是最近一次运行的进程的返回值，所以使用时需要确保在打印需要的进程返回值前没有其他的操作，同时应该避免后台进程的干扰。其次，在运行程序时，如果程序运行出错，比如 shell 找不到指定的进程，那么 \$? 内置变量中的值是 1。

所以，在编写代码时如果没有出错，则不要使 main 函数的返回值为 1，或者使用 exit(1) 这样的写法，以免引起混乱。

(1) 继续回到在 Linux 中如何让一个进程退出（进程退出表示进程即将结束），在 Linux 中进程退出分为正常退出和异常退出两种。

1) 正常退出。

- ①在 main() 函数中执行 return;
- ②调用 exit() 函数;
- ③调用 _exit() 函数。

2) 异常退出。

- ①调用 abort 函数;
- ②进程收到某个信号，而该信号使程序终止。

不管是哪种退出方式，系统最终都会执行内核中的同一代码。这段代码用来关闭进程所用已打开的文件描述符，释放它所占用的内存和其他资源。

(2) 以上几种退出方式的不同点：

1) exit 和 return 的区别。

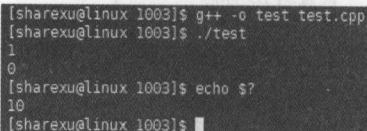
- ① exit 是一个函数，带有参数，exit 执行完后把控制权交给系统;
- ② return 是函数执行完后的返回，return 执行完后把控制权交给调用函数。

2) exit 和 abort 的区别。

- ① exit 是正常终止进程;
- ② abort 是异常终止。

(3) 现在我们重点了解 exit() 和 _exit() 函数。

- 1) exit 和 _exit 函数都是用来终止进程的。



```
[sharexu@linux 1003]$ g++ -o test test.cpp
[sharexu@linux 1003]$ ./test
1
0
[sharexu@linux 1003]$ echo $?
10
[sharexu@linux 1003]$
```

图 10-5 例 10.3 程序的执行结果

当程序执行到 `exit` 或 `_exit` 时，系统无条件地停止剩下所有操作，清除包括 PCB 在内的各种数据结构，并终止本进程的运行。

2) `exit()` 是在头文件 `stdlib.h` 中声明，而 `_exit()` 声明在头文件 `unistd.h` 中声明。`exit` 中的参数 `exit_code` 为 0 时，代表进程正常终止，若为其他值，表示程序执行过程中有错误发生。

并且 `_exit()` 执行后立即返回给内核，而 `exit()` 要先执行一些清除操作，然后将控制权交给内核。

在调用 `_exit` 函数时，它会关闭进程所有的文件描述符，清理内存以及其他一些内核清理函数，但不会刷新流（`stdin`、`stdout`、`stderr`...）的数据。`exit` 函数是在 `_exit` 函数之上的一个封装，其会自动调用 `_exit`，并在调用之前先刷新流数据。

3) `exit()` 函数与 `_exit()` 函数最大区别就在于 `exit()` 函数在调用 `exit` 系统之前要检查文件的打开情况，把文件缓冲区的内容写回文件。由于 Linux 的标准函数库中，有一种被称作“缓冲 I/O”的操作，其特征就是对应每一个打开的文件，在内存中都有一片缓冲区。每次读文件时，会连续的读出若干条记录，这样在下次读文件时就可以直接从内存的缓冲区中读取；同样，每次写文件的时候也仅仅是写入内存的缓冲区，等满足了一定的条件（如达到了一定数量或遇到特定字符等）后，再将缓冲区中的内容一次性写入文件。这种技术大大增加了文件读写的速度，但也给编程带来了一点儿麻烦。比如有一些数据，理论上应该已经写入了文件，但实际上因为没有满足特定的条件，它们还只是保存在缓冲区内，这时如果用 `_exit()` 函数直接将进程关闭，缓冲区的数据就会丢失。因此，要想保证数据的完整性，就一定要使用 `exit()` 函数。

下面通过一个函数实例来看看它们之间的区别。

【例 10.4】`exit` 和 `_exit` 函数的区别。

`exit.cpp` 的代码是：

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    printf("using exit.\n");
    printf("This is the content in buffer");
    exit(0);
}
```

`_exit.cpp` 的代码是：

```
#include <stdio.h>
#include <unistd.h>
int main(){
    printf("using _exit.\n");
    printf("This is the content in buffer");
    _exit(0);
}
```

程序的执行结果如图 10-6 所示。

exit.cpp 和 _exit.cpp 中，只有 exit(0)；和 _exit(0)；不一样；其中的 printf 函数就是使用缓冲 I/O 的方式，该函数在遇到“\n”换行符时自动的从缓冲区中将记录读出。程序执行结果显示，exit() 会将缓冲区的数据写完后才退出，而 _exit() 函数直接退出。

```
[sharexu@linux 1004]$ ./exit
using exit.
This is the content in buffer[sharexu@linux 1004]$ ./_exit
using _exit.
[sharexu@linux 1004]$
```

图 10-6 例 10.4 程序的执行结果

10.3 僵尸进程

假如把例 10.1 中的 sleep(2) 这行代码注释掉，会有可能出现如图 10-7 所示的执行结果。

也就是，子进程在打印父进程 pid 时，发现父进程 pid 的值是 1，和前面打印的父进程 pid 值不一致，这是为什么呢？

```
[sharexu@linux 1001]$ ./test
Parent, PID: 24333, Sub-process PID: 24334
[sharexu@linux 1001]$ Sub-process, PID: 24334, PPID: 1
```

图 10-7 例 10.1 程序去掉休眠 2s 后的执行结果

在 UNIX/Linux 中，正常情况下，子进程是通过父进程创建的，子进程在创建新的进程。子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程到底什么时候结束。于是就产生了孤儿进程和僵尸进程。

孤儿进程，是指一个父进程退出后，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程（进程号为 1）所收养，并由 init 进程对它们完成状态收集工作。

僵尸进程，是指一个进程使用 fork 创建子进程，如果子进程退出，而父进程并没有调用 wait 或 waitpid 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中，这种进程称为僵尸进程。当一个进程完成它的工作终止之后，它的父进程需要调用 wait() 或者 waitpid() 系统调用取得子进程的终止状态。

可以这样理解孤儿进程和僵尸进程的区别：孤儿进程是父进程已退出，而子进程未退出；僵尸进程是父进程未退出，而子进程已退出。

根据上面的描述，可以这样来写一个僵尸进程。

【例 10.5】写一个僵尸进程。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(){
    /*fork 一个子进程*/
    pid_t pid = fork();
    if (pid > 0){/*父进程*/
        printf("in parent process, sleep for one miniute...zZ...\n");
```

```

    sleep(3);
    printf("after sleeping, and exit!\n");
}
else if(pid == 0){
    /* 子进程退出, 成为一个僵尸进程 */
    printf("in child process, and exit!\n");
    exit(0);
}
return 0;
}

```

程序的执行结果如图 10-8 所示。

父进程中休眠 3s, 让子进程先退出。而且, 父进程并没有写 wait 等系统调用函数, 因此在子进程退出之后变成僵尸进程。在程序运行期间, 如果打开另一个终端, 执行 ps aux | grep -w 'Z' 命令, 可以得到如图 10-9 所示的结果。

```

[sharexu@linux 1005]$ ./test
in parent process, sleep for one minute...zZ...
in child process, and exit!
after sleeping, and exit!
[sharexu@linux 1005]$

```

图 10-8 例 10.5 程序的执行结果

```

[sharexu@linux ~]$ ps aux | grep -w 'Z'
sharexu 24666 0.0 0.0 0 0 pts/0 Z+ 20:44 0:00 [test] <defunct>
sharexu 24668 0.0 0.1 103172 932 pts/1 S+ 20:44 0:00 grep -w Z

```

图 10-9 有僵尸进程时用 ps 命令可以看到的结果

从上面可以看出, 系统中多了一个僵尸进程。但如果等父进程睡眠醒来并退出之后, 再次查看系统进程信息, 会发现刚才的僵尸进程不见了, 如图 10-10 所示。因为, 父进程退出后, 这个僵尸进程已成为孤儿进程, 过继给了 init 进程, 而 init 进程会周期性地调用 wait 系统调用来清除各个僵尸的子进程。

```

[sharexu@linux ~]$ ps aux | grep -w 'Z'
sharexu 24670 0.0 0.1 103172 928 pts/1 S+ 20:46 0:00 grep -w Z
[sharexu@linux ~]$

```

图 10-10 没有僵尸进程时用 ps 命令可以看到的结果

既然 wait 函数可以回收子进程, 那 we 来看下怎么使用它。

进程一旦调用了 wait 函数, 就立即阻塞自己, 由 wait 自动分析是否当前进程的某个子进程已经退出, 如果让它找到了这样一个已经变成僵尸的子进程, wait 就会收集这个子进程的信息, 并把它彻底销毁后返回; 如果没有找到这样一个子进程, wait 就会一直阻塞在这里, 直到有一个出现为止。

头文件为:

```

#include<sys/types.h>
#include<sys/wait.h>

```

函数原型为:

```

pid_t wait(int * status);

```


wait() 会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 wait() 时子进程已经结束，则 wait() 会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一并返回。如果不需要结束状态值，则参数 status 可以设成 NULL。

如果执行成功则返回子进程识别码 (PID)；如果有错误发生则返回 -1，并将失败原因存于 errno 中。

【例 10.6】使用 wait 函数回收子进程。

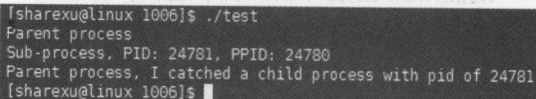
```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(){
    /*fork 一个子进程 */
    pid_t pid = fork();
    if(pid<0){
        perror("fork error\n");
        return 0;
    }else if(pid > 0){/* 父进程 */
        printf("Parent process\n");
        pid_t pr=wait(NULL);
        printf("Parent process, I caught a child process with pid of %d\n",pr);
    }else if(pid == 0){
        printf("Sub-process, PID: %u, PPID: %u\n", getpid(), getppid());
        exit(0);
    }
    return 0;
}
```

程序的执行结果如图 10-11 所示。

例 10.6 中，父进程只负责回收退出的子进程信息，而子进程也只打印了一行提示信息。如果参数 status 的值不是 NULL，wait 就会把子进程退出时的状态取出并存入其中，这是一个整数值 (int)，指出了子

进程是正常退出还是被非正常结束的，以及正常结束时的返回值或被哪一个信号结束的等信息。由于这些信息被存放在一个整数的不同二进制位中，所以用常规的方法读取会非常麻烦，人们就设计了一套专门的宏 (macro) 来完成这项工作，下面介绍其中最常用的两个：① WIFEXITED (status)，这个宏用来指出子进程是否为正常退出的，如果是，它会返回一个非零值，注意，虽然名字一样，这里的参数 status 并不同于 wait 函数中的 status 参数，而是那个指针所指向的整数，切记不要搞混了；② WEXITSTATUS (status)，当 WIFEXITED



```
[sharexu@linux 1006]$ ./test
Parent process
Sub-process, PID: 24781, PPID: 24780
Parent process, I caught a child process with pid of 24781
[sharexu@linux 1006]$
```

图 10-11 例 10.6 程序的执行结果

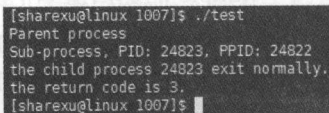
返回非零值时，可以用这个宏来提取子进程的返回值，如果子进程调用 `exit(5)` 退出，`WEXITSTATUS(status)` 就会返回 5，如果子进程调用 `exit(7)`，`WEXITSTATUS(status)` 就会返回 7。请注意，如果进程不是正常退出的，也就是说，`WIFEXITED` 返回 0，这个值就毫无意义。

【例 10.7】 利用 `WEXITSTATUS` 获得子进程的返回码。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(){
    /*fork 一个子进程 */
    pid_t pid = fork();
    if(pid<0){
        perror("fork error\n");
        return 0;
    }else if(pid > 0){/* 父进程 */
        printf("Parent process\n");
        int status=-1;
        pid_t pr=wait(&status);
        if(WIFEXITED(status)){
            printf("the child process %d exit normally.\n",pr);
            printf("the return code is %d.\n",WEXITSTATUS(status));
        }else{
            printf("the child process %d exit abnormally.\n",pr);
        }
    }else if(pid == 0){
        printf("Sub-process, PID: %u, PPID: %u\n", getpid(), getppid());
        exit(3);
    }
    return 0;
}
```

程序的执行结果如图 10-12 所示。

父进程准确捕捉到了子进程的返回值为 3，并把它打印了出来。如果在子进程中再休眠 10s，会发现父进程也在继续等下去，只有子进程从休眠程序中退出，它才能退出，才能父进程捕捉到。读者如果有兴趣的话，可以自己给子进程加上休眠，看看会出现怎样的结果。



```
[sharexu@linux 1007]$ ./test
Parent process
Sub-process, PID: 24823, PPID: 24822
the child process 24823 exit normally.
the return code is 3.
[sharexu@linux 1007]$
```

图 10-12 例 10.7 程序的执行结果

讲到 `wait` 函数，一定会聊到 `waitpid` 函数。从本质上讲，`waitpid` 是 `wait` 的封装，`waitpid` 只是多出了两个可由用户控制的参数 `pid` 和 `options`，为编程提供了灵活性。

使用时必须包括的头文件：

```
#include<sys/types.h>
```

```
#include<sys/wait.h>
```

函数原型：

```
pid_t waitpid(pid_t pid,int * status,int options);
```

waitpid() 会暂时停止目前进程的执行，直到有信号来到或子进程结束。如果在调用 waitpid() 时子进程已经结束，则 waitpid() 会立即返回子进程结束状态值。子进程的结束状态值会由参数 status 返回，而子进程的进程识别码也会一起返回。如果不在意结束状态值，则参数 status 可以设成 NULL。

(1) 参数 pid 为欲等待的子进程识别码，其他数值意义如下所述。

- 1) pid<-1：等待进程组识别码为 pid 绝对值的任何子进程。
- 2) pid=-1：等待任何子进程，相当于 wait。
- 3) pid=0：等待进程组识别码与目前进程相同的任何子进程。
- 4) pid>0：等待任何子进程识别码为 pid 的子进程。

(2) 参数 options 的值有以下几种类型。

1) options=WNOHANG，即使没有子进程退出，它也会立即返回，不会像 wait 那样永远等下去。

2) options=WUNTRACED，则子进程进入暂停则马上返回，但结束状态不予以理会。

Linux 中只支持 WNOHANG 和 WUNTRACED 两个选项，这是两个常数，可以用“|”运算符把它们连接起来使用，比如：

```
ret=waitpid(-1,NULL,WNOHANG | WUNTRACED);
```

如果不想使用它们，也可以把 options 设为 0，如：

```
ret=waitpid(-1,NULL,0);
```

(3) waitpid 的返回值如下所述。

- 1) 当正常返回的时候 waitpid 返回收集到的子进程的进程 ID。
- 2) 如果设置了选项 WNOHANG，而调用中 waitpid 发现没有已退出的子进程可收集，则返回 0。
- 3) 如果调用中出错，则返回 -1，这时 errno 会被设置成相应的值以指示错误所在。
- 4) 当 pid 所指示的子进程不存在，或此进程存在，但不是调用进程的子进程，waitpid 就会出错返回，这时 errno 被设置为 ECHILD。

【例 10.8】 如何使用 waitpid 函数收集子进程信息。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
```

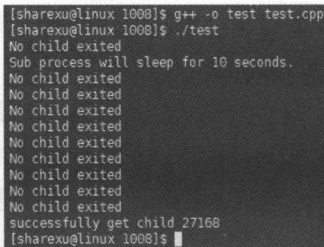
```

#include <stdio.h>
int main(){
    pid_t pid, pr;
    pid=fork();
    if(pid<0) /* 如果 fork 出错 */
        printf("Error occured on forking.\n");
    else if(pid==0){ /* 如果是子进程 */
        printf("Sub process will sleep for 10 seconds.\n");
        sleep(10); /* 睡眠 10 秒 */
        exit(0);
    }else if(pid>0){
        /* 如果是父进程 */
        do{
            pr=waitpid(pid, NULL, WNOHANG);
            /* 使用了 WNOHANG 参数, waitpid 不会在这里等待 */
            if(pr==0){ /* 如果没有收集到子进程 */
                printf("No child exited\n");
                sleep(1);
            }
        }while(pr==0); /* 没有收集到子进程, 就回去继续尝试 */
        if(pr==pid)
            printf("successfully get child %d\n", pr);
        else
            printf("some error occured\n");
    }
    return 0;
}

```

程序的执行结果如图 10-13 所示。

父进程经过 10 次失败的尝试之后, 终于收集到了退出的子进程。因为这只是一个例子程序, 不便写得太复杂, 所以就让父进程和子进程分别休眠 10s 和 1s, 代表它们分别作了 10s 和 1s 的工作。父子进程都有工作要做, 父进程利用工作的简短间歇查看子进程的是否退出, 如退出就收集它。



```

[sharex@linux 1008]$ g++ -o test test.cpp
[sharex@linux 1008]$ ./test
No child exited
Sub process will sleep for 10 seconds.
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
No child exited
successfully get child 27168
[sharex@linux 1008]$

```

图 10-13 例 10.8 程序的执行结果

10.4 守护进程

在 Linux 或者 UNIX 操作系统中在系统的引导的时候会开启很多服务, 这些服务就叫作守护进程。为了增加灵活性, root 可以选择系统开启的模式, 这些模式叫作运行级别, 每一种运行级别以一定的方式配置系统。

守护进程是脱离于终端并且在后台运行的进程。守护进程脱离于终端是为了避免进程在执行过程中的信息在任何终端上显示并且进程也不会被任何终端所产生的终端信息所打断。

守护进程是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。守护进程常常在系统引导装入时启动，在系统关闭时终止。Linux 系统有很多守护进程，大多数服务都是通过守护进程实现的，同时，守护进程还能完成许多系统任务，例如作业规划进程 `crond`、打印进程 `lpd` 等（这里的结尾字母 `d` 就是 `Daemon` 的意思）。

由于在 Linux 中，每一个系统与用户进行交流的界面称为终端，每一个从此终端开始运行的进程都会依附于这个终端，这个终端就称为这些进程的控制终端，当控制终端被关闭时，相应的进程都会自动关闭。但是守护进程却能够突破这种限制，它从被执行时开始运转，直到整个系统关闭时才退出。如果想让某个进程不因为用户或终端或其他地变化而受到影响，那么就必须把这个进程变成一个守护进程。

创建一个简单的守护进程的步骤如下所述。

（1）创建子进程，父进程退出。

这是编写守护进程的第一步。由于守护进程是脱离控制终端的，因此完成第一步后就会在 Shell 终端里造成一程序已经运行完毕的假象。之后的所有工作都在子进程中完成，而用户在 Shell 终端里则可以执行其他命令，从而在形式上做到了与控制终端的脱离。

在 Linux 中如果父进程先于子进程退出会造成子进程成为孤儿进程，而每当系统发现一个孤儿进程时，就会自动由 1 号进程（`init`）收养它，这样，原先的子进程就会变成 `init` 进程的子进程。

（2）在子进程中创建新会话。

这个步骤是创建守护进程中最重要的一步，虽然它的实现非常简单，但它的意义却非常重大。在这里使用的是系统函数 `setsid`，在具体介绍 `setsid` 之前，首先要了解两个概念：进程组和会话期。

1) 进程组：是一个或多个进程的集合。进程组由进程组 ID 来唯一标识，除了进程号（`PID`）之外，进程组 ID 也是一个进程的必备属性。每个进程组都有一个组长进程，其组长进程的进程号等于进程组 ID，且该进程组 ID 不会因组长进程的退出而受到影响。

2) 会话周期：会话期是一个或多个进程组的集合。通常，一个会话开始与用户登录，终止于用户退出，在此期间该用户运行的所有进程都属于这个会话期。

`setsid` 函数用于创建一个新的会话，并担任该会话组的组长。调用 `setsid` 有 3 个作用：①让进程摆脱原会话的控制；②让进程摆脱原进程组的控制；③让进程摆脱原控制终端的控制。

那么，在创建守护进程时为什么要调用 `setsid` 函数呢？由于创建守护进程的第一步调用了 `fork` 函数来创建子进程，再将父进程退出。由于在调用了 `fork` 函数时，子进程全盘拷贝了父进程的会话期、进程组、控制终端等，虽然父进程退出了，但会话期、进程组、控制终

端等并没有改变，因此，还不是真正意义上的独立。而 `setsid` 函数能够使进程完全独立出来，从而摆脱其他进程的控制。

(3) 改变当前目录为根目录。

这一步也是必要的步骤。使用 `fork` 创建的子进程继承了父进程的当前工作目录。由于在进程运行中，当前目录所在的文件系统（如 `/mnt/usb`）是不能卸载的，这对以后的使用会造成诸多的麻烦（比如系统由于某种原因要进入用户模式，但无法实现）。因此，通常的做法是让 `"/"` 作为守护进程的当前工作目录，这样就可以避免上述的问题。当然，如有特殊需要，也可以把当前工作目录换成其他的路径。

(4) 重设文件权限掩码。

文件权限掩码是指屏蔽掉文件权限中的对应位。比如，有个文件权限掩码是 `050`，它就屏蔽了文件组拥有者的可读与可执行权限。由于使用 `fork` 函数新建的子进程继承了父进程的文件权限掩码，这就给该子进程使用文件带来了诸多的麻烦。因此，把文件权限掩码设置为 `0`，可以大大增强该守护进程的灵活性。设置文件权限掩码的函数是 `umask`，通常的使用方法是 `umask(0)`。

(5) 关闭文件描述符。

同文件权限码一样，用 `fork` 函数新建的子进程会从父进程那里继承一些已经打开了的文件。这些被打开的文件可能永远不会被守护进程读写，但它们一样消耗系统资源，而且可能导致所在的文件系统无法结束。

在上面的第二步之后，守护进程已经与所属的控制终端失去了联系。因此从终端输入的字符不可能达到守护进程，守护进程中用常规方法输出的字符（如 `printf`）也不可能在终端上显示出来。所以，文件描述符为 `0`、`1` 和 `2` 的 3 个文件（常说的输入、输出和报错）已经失去了存在的价值，也应被关闭。通常按如下方式关闭文件描述符：

```
for(i=0;i<MAXFILE;i++)
close(i);
```

这样，一个简单的守护进程就建立起来了。

【例 10.9】 实现一个守护进程的完整实例（每隔 10s 在 `/tmp/dameon.log` 中写入一句话）。

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<fcntl.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>
#include<sys/stat.h>
```

```

#define MAXFILE 65535
int main(){
    pid_t pc;
    int i,fd,len;
    char *buf="this is a Dameon\n";
    len = strlen(buf);
    pc = fork(); /* 第一步 */
    if(pc<0){
        printf("error fork\n");
        exit(1);
    }else if(pc>0){
        exit(0);
    }
    setsid(); /* 第二步 */
    chdir("/"); /* 第三步 */
    umask(0); /* 第四步 */
    for(i=0;i<MAXFILE;i++) /* 第五步 */
        close(i);
    while(1){
        if((fd=open("/tmp/dameon.log",O_CREAT|O_WRONLY|O_APPEND,0600))<0){
            perror("open");
            exit(1);
        }
        write(fd,buf,len+1);
        close(fd);
        sleep(10);
    }
    return 0;
}

```

执行编译后的目标文件，发现它好像执行一下就退出了，如图 10-14 所示。

这时，执行 `ps -ef | grep test` 命令，发现它还在运行，只是躲到后台去运行了，如图 10-15 所示。

```

[sharexu@linux 1009]$ g++ -o test test.cpp
[sharexu@linux 1009]$ ./test
[sharexu@linux 1009]$

```

图 10-14 例 10.9 程序的执行结果

```

[sharexu@linux 1009]$ ps -ef | grep test
sharexu 27970 1 0 21:20 ? 00:00:00 ./test
sharexu 27972 27832 0 21:21 pts/0 00:00:00 grep test
[sharexu@linux 1009]$

```

图 10-15 用 ps 命令发现后台运行的程序

再去查看 `/tmp/dameon.log` 文件，还在每隔 10s 就打印一行 “this is a Dameon”，如图 10-16 所示。

如果不想这个 dameon 程序继续进行了，可以直接把进程杀掉，这样进程就可以结束了，如图 10-17 所示。


```
[sharexu@linux 1009]$ tail /tmp/dameon.log
this is a Dameon
this is a Dameon
this is a Dameon
this is a Dameon
this is a Dameon
this is a Dameon
this is a Dameon
this is a Dameon
this is a Dameon
[sharexu@linux 1009]$
```

图 10-16 例 10.8 程序修改的文件还在不停地被修改

```
[sharexu@linux 1009]$ kill -9 27970
[sharexu@linux 1009]$ ps -ef | grep test
sharexu 27977 27832 0 21:24 pts/0 00:00:00 grep test
[sharexu@linux 1009]$
```

图 10-17 杀掉进程后, 用 ps 命令就看不到该程序在执行了

所以, 如果你想创建一个可以脱离终端运行的进程, 守护进程是优秀的选择。

10.5 本章小结

本章主要介绍了进程的创建和使用, 还介绍了孤儿进程、僵尸进程和守护进程的内存及使用。接下来的第 11 章, 将介绍进程间如何通信的问题。

进程间通信

进程间通信就是不同进程之间传播或交换信息，那么不同进程之间存在着什么双方都可以访问的介质呢？首先，进程间通信至少可以通过传送、打开文件来实现，不同的进程通过一个或多个文件来传递信息，事实上，在很多应用系统里都使用了这种方法。但一般说来，进程间通信（Inter Process Communication, IPC）不包括这种似乎比较低级的通信方法。UNIX 系统中实现进程间通信的方法很多，而且不幸的是，极少方法能在所有的 UNIX 系统中进行移植（唯一一种是半双工的管道，这也是最原始的一种通信方式）。而 Linux 作为一种新兴的操作系统，几乎支持所有的 UNIX 下常用的进程间通信方法：管道、消息队列、共享内存、信号量、套接字等。其中，前面 4 种主要用于同一台机器上的进程间通信，而套接字则主要用于不同机器之间的网络通信。由于第 6 章已介绍了套接字编程，本章不再赘述，将主要介绍前 4 种方式。

11.1 管道

第 10 章中已经介绍了父子进程之间并不共享数据段和堆栈段，它们之间是通过管道进行通信的。管道是一种两个进程间进行单向通信的机制。因为管道传递数据的单向性，管道又称为半双工管道，管道的这一特点决定了其使用的局限性。管道是 Linux 支持的最初 UNIX IPC 形式之一，具有以下特点。

（1）数据只能由一个进程流向另一个进程（其中一个读管道，一个写管道）；如果要进行双工通信，则需要建立两个管道。

（2）管道只能用于父子进程或者兄弟进程间通信，也就是说管道只能用于具有亲缘关系

的进程间通信。

除了以上局限性,管道还有其他一些不足,如管道没有名字(无名管道);管道的缓冲区大小是受限制的;管道所传输的是无格式的字节流等。这就需要管道输入方和输出方事先约定好数据格式。虽然有那么多不足,但对于一些简单的进程间通信,管道还是完全可以胜任的。

使用管道进行通信时,两端的进程向管道读写数据是通过创建管道时,系统设置的文件描述符进行的。从本质上说,管道也是一种文件,但它又和一般的文件有所不同,可以克服使用文件进行通信的两个问题,这个文件只存在内存中。

通过管道通信的两个进程,一个进程向管道写数据,另外一个从中读数据。写入的数据每次都添加到管道缓冲区的末尾,读数据的时候都是从缓冲区的头部读出数据的。

管道由 `pipe()` 函数创建,需要依赖的头文件是:

```
#include <unistd.h>
```

`pipe()` 的函数原型是:

```
int pipe(int fd[2]);
```

`pipe()` 函数创建的管道处于一个进程中间,在实际应用中没有太大的意义。因此,一个进程在由 `pipe()` 创建管道后,一般再使用 `fork` 建立一个子进程,然后通过管道实现父子进程间的通信。管道两端可分别用描述字 `fd[0]` 以及 `fd[1]` 来描述。需要注意的是,管道的两端是固定了任务的,即一端只能用于读,由描述字 `fd[0]` 表示,称其为管道读端;另一端则只能用于写,由描述字 `fd[1]` 来表示,称其为管道写端。如果试图从管道写端读取数据,或者向管道读端写入数据都将导致错误发生。一般文件的 I/O 函数都可以用于管道,如 `close`、`read`、`write` 等。

例 11.1 示范了如何在父进程和子进程间实现通信。

【例 11.1】 利用管道实现在父子进程间通信。

```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define INPUT 0
#define OUTPUT 1
int main() {
    int fd[2];
    /* 定义子进程号 */
    pid_t pid;
    char buf[256];
    int returned_count;
    /* 创建无名管道 */
    pipe(fd);
    /* 创建子进程 */
    pid=fork();
```

```

if(pid<0) {
    printf("Error in fork\n");
    exit(1);
}else if(pid == 0) { /* 执行子进程 */
    printf("in the child process...\n");
    /* 子进程向父进程写数据, 关闭管道的读端 */
    close(fd[INPUT]);
    write(fd[OUTPUT], "hello world", strlen("hello world"));
    exit(0);
}else { /* 执行父进程 */
    printf("in the parent process...\n");
    /* 父进程从管道读取子进程写的的数据, 关闭管道的写端 */
    close(fd[OUTPUT]);
    returned_count = read(fd[INPUT], buf, sizeof(buf));
    printf("%d bytes of data received from child process: %s\n", returned_
count, buf);
}
return 0;
}

```

程序的执行结果是:

```

in the parent process...
in the child process...
11 bytes of data received from child process: hello world

```

例 11.1 中, 在子进程中写数据, 在父进程中读数据, 两个进程之间实现了通信: 父子进程分别拥有自己的读写通道, 为了实现父子进程之间的读写, 只需把无关的读端或写端的文件描述符关闭即可。例 11.1 中, 将父进程的写端 `fd[1]` 和子进程的读端 `fd[0]` 关闭, 则父子进程之间就建立起一条“子进程写入父进程读取”的通道。同样, 也可以将父进程的读端 `fd[0]` 和子进程的写端 `fd[1]` 关闭, 则父子进程之间就建立起一条“父进程写入子进程读取”的通道; 上述管道又被称为无名管道。

还有一种管道叫有名管道 (named pipe 或 FIFO), 它不同于无名管道之处在于它提供一个路径名与之关联, 以 FIFO 的文件形式存在于文件系统中。这样, 即使与 FIFO 的创建进程不存在亲缘关系的进程, 只要可以访问该路径, 就能够彼此通过 FIFO 相互通信 (能够访问该路径的进程以及 FIFO 的创建进程之间), 因此, 通过 FIFO 不相关的进程也能交换数据。

有名管道是对无名管道的一种改进, 二者区别如图 11-1 所示。

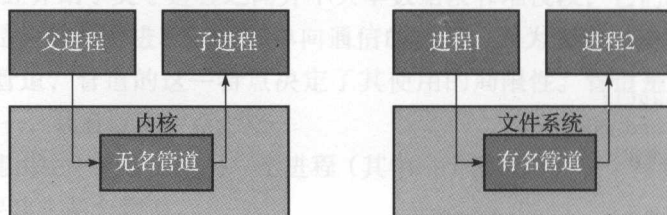


图 11-1 有名管道与无名管道的区别

有名管道具有以下特点：①它可以使互不相关的两个进程间实现彼此通信；②该管道可以通过路径名来指出，并且在文件系统中是可见的。在建立了管道之后，两个进程就可以把它当作普通文件一样进行读写操作，使用非常方便；③ FIFO 严格地遵循先进先出规则，对管道及 FIFO 的读操作总是从开始处返回数据，对它们的写操作则是把数据添加到末尾。

有名管道由 `mkfifo()` 函数创建，需要依赖的头文件是：

```
#include <sys/types.h>
#include <sys/stat.h>
```

`mkfifo()` 的函数原型是：

```
int mkfifo(const char * pathname, mode_t mode)
```

该函数的第一个参数是一个普通的路径名，也就是创建后 FIFO 的名字。第二个参数与打开普通文件的 `open()` 函数中的 `mode` 参数相同。如果 `mkfifo` 的第一个参数是一个已经存在的路径名时，会返回 `EEXIST` 错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开 FIFO 的函数就可以了。一般文件的 I/O 函数都可以用于 FIFO，如 `close`、`read`、`write` 等。

【例 11.2】两个进程用有名管道进行通信。

功能：一共有两个程序。一个程序用于读管道，另一个程序用于写管道。其中在读管道的程序中创建管道，并且读出用户写入到管道的内容；写管道的程序则把 `main()` 函数里的参数写入管道中。这两个程序采用的是非阻塞式读写管道模式。

读管道程序 `mkfifo_r.cpp` 的源码如下：

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#define P_FIFO "/tmp/p_fifo"
int main(int argc, char** argv){
    char cache[100];
    int fd;
    memset(cache,0, sizeof(cache)); /* 初始化内存 */
    if(access(P_FIFO,F_OK)==0){ /* 管道文件存在 */
        execlp("rm","-f", P_FIFO, NULL); /* 删掉 */
        printf("access.\n");
    }
    if(mkfifo(P_FIFO, 0777) < 0){
        printf("createnamed pipe failed.\n");
    }
    fd= open(P_FIFO,O_RDONLY|O_NONBLOCK); /* 非阻塞方式打开，只读 */
    while(1){
        memset(cache,0, sizeof(cache));
```

```

        if((read(fd,cache, 100)) == 0 ){ /* 没有读到数据 */
            printf("nodata:\n");
        }
        else{
            printf("getdata:%s\n", cache); /* 读到数据, 将其打印 */
        }
        sleep(1); /* 休眠 1s */
    }
    close(fd);
    return 0;
}

```

写管道程序 mkfifo_w.cpp 的源码如下:

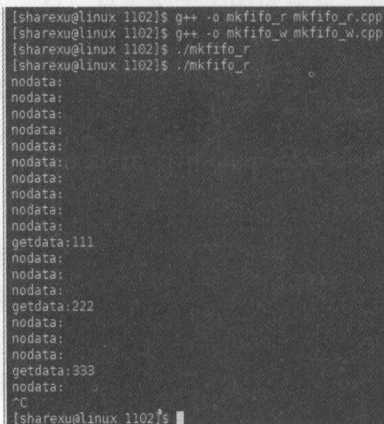
```

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#define P_FIFO "/tmp/p_fifo"
int main(int argc, char **argv){
    int fd;
    if(argc < 2){
        printf("please input the write data.\n");
    }
    fd= open(P_FIFO,O_WRONLY|O_NONBLOCK); /* 非阻塞方式 */
    write(fd,argv[1], 100); /* 将 argv[1] 写道 fd 里面去 */
    close(fd);
    return 0;
}

```

编写保存上述两个文件后分别使用命令: `g++ -o mkfifo_r mkfifo_r.cpp` 和命令: `g++ -o mkfifo_w mkfifo_w.cpp` 编译。

为了更好地观察运行效果,需要把这两个程序分别在终端里运行,在这里首先启动读管道程序。读管道进程在建立管道后就开始循环地从管道里读出内容,如果没有数据可读,则一直阻塞到写管道进程向管道写入数据。在启动了写管道程序后,读进程能够从管道里读出用户的输入内容,程序运行结果如图 11-2、图 11-3、图 11-4 所示。

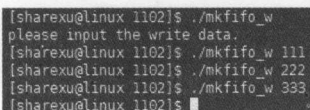


```

[sharexu@linux 1102]$ g++ -o mkfifo_r mkfifo_r.cpp
[sharexu@linux 1102]$ g++ -o mkfifo_w mkfifo_w.cpp
[sharexu@linux 1102]$ ./mkfifo_r
nodata:
nodata:
nodata:
nodata:
nodata:
nodata:
nodata:
nodata:
nodata:
getdata:111
nodata:
nodata:
getdata:222
nodata:
nodata:
nodata:
getdata:333
nodata:
^C
[sharexu@linux 1102]$

```

图 11-2 例 11.2 读进程执行结果截图

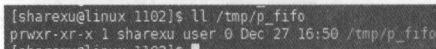


```

[sharexu@linux 1102]$ ./mkfifo_w
please input the write data.
[sharexu@linux 1102]$ ./mkfifo_w 111
[sharexu@linux 1102]$ ./mkfifo_w 222
[sharexu@linux 1102]$ ./mkfifo_w 333
[sharexu@linux 1102]$

```

图 11-3 例 11.2 写进程执行结果截图



```

[sharexu@linux 1102]$ ll /tmp/p_fifo
prwxr-xr-x 1 sharexu user 0 Dec 27 16:50 /tmp/p_fifo
[sharexu@linux 1102]$

```

图 11-4 例 11.2 生成的文件

mkfifo_r 和 mkfifo_w 通过有名管道进行了通信, mkfifo_w 发送了什么内容, mkfifo_r 就将收到了什么内容。

再来看一下详细的代码。读管道程序 mkfifo_r.cpp 中, 先判断管道文件是否存在, 如果存在, 先把它删掉。

```
if(access(P_FIFO,F_OK)==0){ /* 管道文件存在 */
    execlp("rm","-f", P_FIFO, NULL); /* 删掉 */
    printf("access.\n");
}
```

创建一个有名管道, 注意判断是否创建成功, 代码如下:

```
if(mkfifo(P_FIFO, 0777) < 0){
    printf("createnamed pipe failed.\n");
}
```

以只读方式打开有名管道, 代码如下:

```
fd= open(P_FIFO,O_RDONLY|O_NONBLOCK); /* 非阻塞方式打开, 只读 */
```

每隔 1s 就去有名管道中读数据, 如果读到了数据, 将其打印, 如果没有读到数据, 则输出 nodata。注意, 读数据之前, 要将缓存清空, 以免被上一次的读取结果影响, 代码如下:

```
while(1){
    memset(cache,0, sizeof(cache));
    if((read(fd,cache, 100)) == 0 ){ /* 没有读到数据 */
        printf("nodata:\n");
    }
    else{
        printf("getdata:%s\n", cache); /* 读到数据, 将其打印 */
    }
    sleep(1);/* 休眠 1s*/
}
```

写管道程序 mkfifo_w.cpp 中, 只要先以可写的方式打开有名管道, 并往里头写数据即可, 代码如下:

```
fd= open(P_FIFO,O_WRONLY|O_NONBLOCK); /* 非阻塞方式 */
write(fd,argv[1], 100); /* 将 argv[1] 写道 fd 里面去 */
```

你也许会有这样的疑问, 能否使用有名管道让一个服务器和多个客户端进行双向交流? 事实上, 一对多的形式经常出现, 只要每次客户端向服务器发出的指令小于 PIPE_BUF (管道写入最大值), 它们就可以通过一个有名管道向服务器发送数据, 并且客户端可以很容易地知道服务器传发数据的管道名。但问题在于, 服务器不能用一个管道来和所有客户打交道。如果不止一个客户在读同一个管道, 则无法确保每个客户都得到自己对应的回复。

一个解决办法就是每个客户在向服务器发送信息前都建立自己的读入管道, 或让服务器在得到数据后再建立管道。使用客户的进程号 (pid) 作为管道名是一种常用的方法。客户可以先把自己的进程号告诉服务器, 然后到那个以自己进程号命名的管道中读取回复。

11.2 消息队列

消息队列用于运行于同一台机器上的进程间通信，它和管道很相似，是一个在系统内核中用来保存消息的队列，它在系统内核中是以消息链表的形式出现。消息链表中节点的结构用 msg 声明。

相关的函数有以下几个。

(1) 创建新消息队列或取得已存在消息队列，函数原型是：

```
int msgget(key_t key, int msgflg);
```

参数中：① key 可以认为是一个端口号，也可以由函数 ftok 生成。② msgflg 如果等于 IPC_CREAT，若没有该队列，则创建一个并返回新标识符，若已存在则返回原标识符；msgflg 如果等于 IPC_EXCL，若没有该队列，则返回 -1；若已存在，则返回 0；

(2) 向队列读 / 写消息，函数原型如下所述。

msgrcv 从队列中取用消息：

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

msgsnd 将数据放到消息队列中：

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

参数中：① msqid 是消息队列的标识码；② msgp 是指向消息缓冲区的指针。此位置用来暂时存储发送和接收的消息，是一个用户可定义的通用结构，但一般定义为以下结构：

```
struct msgstru{
    long mtype;    // 大于 0
    char mtext[512];
};
```

msgsz 是指消息的大小；msgtyp 是指从消息队列内读取的消息形态。如果值为零，则表示消息队列中的所有消息都会被读取。

msgflg：用来指明核心程序在队列没有数据的情况下所应采取的行动。如果 msgflg 和常数 IPC_NOWAIT 合用，则在 msgsnd() 执行时若是消息队列已满，则 msgsnd() 将不会阻塞，而会立即返回 -1，如果执行的是 msgrcv()，则在消息队列呈空时，不做等待马上返回 -1，并设定错误码为 ENOMSG。当 msgflg 为 0 时，msgsnd() 及 msgrcv() 在队列呈满或呈空的情形时，采取阻塞等待的处理模式。

(3) 设置消息队列属性，函数原型是：

```
int msgctl(int msgqid, int cmd, struct msgid_ds *buf);
```

参数中的 msgctl 系统调用对 msgqid 标识的消息队列执行 cmd 操作，系统定义了 3 种 cmd 操作：IPC_STAT、IPC_SET、IPC_RMID。IPC_STAT 用来获取消息队列对应的 msgid_ds 数据

结构,并将其保存到 buf 指定的地址空间;IPC_SET 用来设置消息队列的属性,要设置的属性存储在 buf 中;IPC_RMID 用来从内核中删除 msgid 标识的消息队列。

下面用实例来看下如何用消息队列来进行进程间的通信。

【例 11.3】用消息队列来传输数据。

接收消息 msgreceive.cpp 的源码如下:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/msg.h>
struct msg_st{
    long int msg_type;
    char text[BUFSIZ];
};
int main(){
    int running = 1;
    int msgid = -1;
    struct msg_st data;
    long int msgtype = 0;

    /* 建立消息队列 */
    msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
    if(msgid == -1){
        fprintf(stderr, "msgget failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }
    /* 从队列中获取消息,直到遇到 end 消息为止 */
    while(running){
        if(msgrcv(msgid, (void*)&data, BUFSIZ, msgtype, 0) == -1){
            fprintf(stderr, "msgrcv failed with errno: %d\n", errno);
            exit(EXIT_FAILURE);
        }
        printf("You wrote: %s\n",data.text);
        /* 遇到 end 结束 */
        if(strncmp(data.text, "end", 3) == 0){
            running = 0;
        }
    }
    /* 删除消息队列 */
    if(msgctl(msgid, IPC_RMID, 0) == -1){
        fprintf(stderr, "msgctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

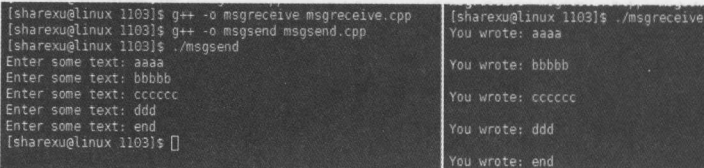
发送消息 msgsend.cpp 的源码如下:

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/msg.h>
#include <errno.h>
#define MAX_TEXT 512
struct msg_st{
    long int msg_type;
    char text[MAX_TEXT];
};
int main(){
    int running = 1;
    struct msg_st data;
    char buffer[BUFSIZ];
    int msgid = -1;
    /* 建立消息队列 */
    msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
    if(msgid == -1){
        fprintf(stderr, "msgget failed with error: %d\n", errno);
        exit(EXIT_FAILURE);
    }
    /* 向消息队列中写消息, 直到写入 end */
    while(running){
        /* 输入数据 */
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);
        data.msg_type = 1;
        strcpy(data.text, buffer);
        /* 向队列发送数据 */
        if(msgsnd(msgid, (void*)&data, MAX_TEXT, 0) == -1){
            fprintf(stderr, "msgsnd failed\n");
            exit(EXIT_FAILURE);
        }
        /* 输入 end 结束输入 */
        if(strncmp(buffer, "end", 3) == 0)
            running = 0;
        sleep(1);
    }
    exit(EXIT_SUCCESS);
}

```

程序的执行结果如图 11-5 所示。



```

[sharexu@linux 1103]$ g++ -o msgreceive msgreceive.cpp
[sharexu@linux 1103]$ g++ -o msgsend msgsend.cpp
[sharexu@linux 1103]$ ./msgsend
Enter some text: aaaa
Enter some text: bbbbbb
Enter some text: cccccc
Enter some text: ddd
Enter some text: end
[sharexu@linux 1103]$

[sharexu@linux 1103]$ ./msgreceive
You wrote: aaaa
You wrote: bbbbbb
You wrote: cccccc
You wrote: ddd
You wrote: end

```

图 11-5 例 11.3 程序执行结果图

注意 msgreceive.cpp 文件 main 函数中定义的变量 msgtype，它作为 msgrcv 函数的接收消息类型参数的值，其值为 0，表示获取队列中第一个可用的消息。再来看看 msgsend.cpp 文件中 while 循环中的语句 data.msg_type=1，它用来设置发送的消息类型，即其发送的消息的类型为 1。所以程序 msgreceive 能够接收到程序 msgsend 发送的消息。下面再来看下程序详情。

msgreceive.cpp 和 msgsend.cpp 中，都需要建立消息队列，不像有名管道，只需要一方建立有名管道即可，代码如下：

```
/* 建立消息队列 */
msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
if(msgid == -1){
    fprintf(stderr, "msgget failed with error: %d\n", errno);
    exit(EXIT_FAILURE);
}
```

msgreceive.cpp 中使用 msgrcv 从队列中获取消息，直到遇到 end 消息为止，代码如下：

```
while(running){
    if(msgrcv(msgid, (void*)&data, BUFSIZ, msgtype, 0) == -1){
        fprintf(stderr, "msgrcv failed with errno: %d\n", errno);
        exit(EXIT_FAILURE);
    }
    printf("You wrote: %s\n", data.text);
    /* 遇到 end 结束 */
    if(strncmp(data.text, "end", 3) == 0){
        running = 0;
    }
}
```

程序结束时，要删除消息队列，代码如下：

```
if(msgctl(msgid, IPC_RMID, 0) == -1){
    fprintf(stderr, "msgctl(IPC_RMID) failed\n");
    exit(EXIT_FAILURE);
}
```

msgsend.cpp 中，用 msgsnd 发送消息，直到需要发送的内容是 end 为止，代码如下：

```
/* 向消息队列中写消息，直到写入 end */
while(running){
    /* 输入数据 */
    printf("Enter some text: ");
    fgets(buffer, BUFSIZ, stdin);
    data.msg_type = 1;
    strcpy(data.text, buffer);
    /* 向队列发送数据 */
    if(msgsnd(msgid, (void*)&data, MAX_TEXT, 0) == -1){
```

```

    fprintf(stderr, "msgsnd failed\n");
    exit(EXIT_FAILURE);
}
/* 输入end 结束输入 */
if(strncmp(buffer, "end", 3) == 0)
    running = 0;
    sleep(1);
}

```

消息队列跟有名管道有不少的相同之处，消息队列进行通信的进程可以是不相关的进程，同时它们都是通过发送和接收的方式来传递数据的。在命名管道中，发送数据用 `write` 函数，接收数据用 `read` 函数，则在消息队列中，发送数据用 `msgsnd` 函数，接收数据用 `msgrcv` 函数。而且它们对每个数据都有一个最大长度的限制。

与命名管道相比，消息队列的优势在于：①消息队列也可以独立于发送和接收进程而存在，从而消除了在同步命名管道的打开和关闭时可能产生的困难；②可以同时通过发送消息以避免命名管道的同步和阻塞问题，而不需要由进程自己来提供同步方法；③接收程序可以通过消息类型有选择地接收数据，而不是像命名管道中那样，只能默认地接收。

事实上，它是一种正逐渐被淘汰的通信方式，完全可以用流管道或者套接口的方式来取代它，所以，建议读者忽略这种方式。

11.3 共享内存

顾名思义，共享内存就是允许两个不相关的进程访问同一个逻辑内存。共享内存是在两个正在运行的进程之间共享和传递数据的一种非常有效的方式。不同进程之间共享的内存通常安排在同一段物理内存中。进程可以将同一段共享内存连接到它们自己的地址空间中，所有进程都可以访问共享内存中的地址，就好像它们是由用 C 语言函数 `malloc` 分配的内存一样。而如果某个进程向共享内存写入数据，所做的改动将立即影响到可以访问同一段共享内存的任何其他进程。

不过，共享内存并未提供同步机制，也就是说，在第一个进程对共享内存的写操作结束之前，并无自动机制可以阻止第二个进程对它进行读取。所以通常需要用其他的机制来同步对共享内存的访问。

在 Linux 中也提供了一组函数接口用于使用共享内存，首先常用的函数是 `shmget`，该函数用来创建共享内存，它用到的头文件是：

```
#include <sys/shm.h>
```

函数原型是：

```
int shmget(key_t key, int size, int flag);
```


第一个参数，程序需要提供一个参数 `key`（非 0 整数），它有效地为共享内存段命名，`shmget` 函数运行成功时会返回一个与 `key` 相关的共享内存标识符（非负整数），用于后续的共享内存函数；调用失败返回 -1。

不相关的进程可以通过该函数的返回值访问同一共享内存，它代表程序可能要使用的某个资源，程序对所有共享内存的访问都是间接的。程序先通过调用 `shmget` 函数并提供一个键，再由系统生成一个相应的共享内存标识符（`shmget` 函数的返回值）。

第二个参数，`size` 以字节为单位指定需要共享的内存容量。

第三个参数，`shmflg` 是权限标志，它的作用与 `open` 函数的 `mode` 参数一样，如果要想在 `key` 标识的共享内存不存在的条件下创建它的话，可以与 `IPC_CREAT` 做或操作。共享内存的权限标志与文件的读写权限一样，举例来说，0644 表示允许一个进程创建的共享内存被内存创建者所拥有的进程向共享内存读取和写入数据，同时其他用户创建的进程只能读取共享内存。

当共享内存创建后，其余进程可以调用 `shmat` 将其连接到自身的地址空间中，它的函数原型是：

```
void *shmat(int shmid, void *addr, int flag);
```

`shmid` 为 `shmget` 函数返回的共享存储标识符，`addr` 和 `flag` 参数决定了以什么方式来确定连接的地址，函数的返回值即是该进程数据段所连接的实际地址，其他进程可以对此进程进行读写操作。

`shmdt` 函数用于将共享内存从当前进程中分离。注意，将共享内存分离并不是删除它，只是使该共享内存对当前进程不再可用。它的原型如下：

```
int shmdt(const void *shmaddr);
```

参数 `shmaddr` 是 `shmat` 函数返回的地址指针，调用成功时返回 0，失败时返回 -1。

例 11.4 就以两个不相关的进程来说明进程间如何通过共享内存来进行通信。其中一个文件 `consumer.cpp` 创建共享内存，并读取其中的信息，另一个文件 `producer.cpp` 向共享内存中写入数据。为了方便操作和数据结构的统一，在文件 `shm_com.h` 中为这两个文件定义了相同的数据结构。结构 `shared_use_st` 中的 `written` 作为一个可读或可写的标志，非 0 表示可读，0 表示可写，`text` 则是内存中的文件。

【例 11.4】使用共享内存进行进程间通信。

`shm_com.h` 的源码如下：

```
#ifndef _SHMCOM_H_HEADER
#define _SHMCOM_H_HEADER
#define TEXT_SZ 2048
struct shared_use_st {
    int written;
```

```

char text[TEXT_SZ];
};
#endif

```

consumer.cpp 的源码如下:

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/shm.h>
#include "shm_com.h"

int main(){
    int shmid;
    srand((unsigned int) getpid());
    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }
    void *shared_memory = (void *)0;
    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Memory attached at %X\n", (long)shared_memory);
    struct shared_use_st *shared_stuff;
    shared_stuff = (struct shared_use_st *)shared_memory;
    shared_stuff->written = 0;
    int running = 1;
    while(running){
        if (shared_stuff->written){
            printf("You wrote: %s", shared_stuff->text);
            sleep( rand() % 4 );
            shared_stuff->written = 0;
            if (strncmp(shared_stuff->text, "end", 3) == 0) {
                running = 0;
            }
        }
    }
    if (shmdt(shared_memory) == -1){
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }
    if (shmctl(shmid, IPC_RMID, 0) == -1){
        fprintf(stderr, "shmctl(IPC_RMID) failed\n");
        exit(EXIT_FAILURE);
    }
}

```

```
exit(EXIT_SUCCESS);
}
```

producer.cpp 的源代码如下:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/shm.h>
#include "shm_com.h"

int main(){
    int shmid;
    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);
    if (shmid == -1){
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }
    void *shared_memory = (void *)0;
    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1){
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }
    printf("Memory attached at %X\n", (long)shared_memory);
    struct shared_use_st *shared_stuff;
    shared_stuff = (struct shared_use_st *)shared_memory;
    int running = 1;
    char buffer[BUFSIZ];
    while(running){
        while(shared_stuff->written == 1){
            sleep(1);
            printf("waiting for client...\n");
        }
        printf("Enter some text: ");
        fgets(buffer, BUFSIZ, stdin);
        strncpy(shared_stuff->text, buffer, TEXT_SZ);
        shared_stuff->written = 1;
        if (strcmp(buffer, "end", 3) == 0) {
            running = 0;
        }
    }
    if (shmdt(shared_memory) == -1) {
        fprintf(stderr, "shmdt failed\n");
        exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

程序的执行结果如图 11-6 和图 11-7 所示。

```

[sharexu@linux 1104]$ g++ -o consumer consumer.cpp
[sharexu@linux 1104]$ g++ -o producer producer.cpp
[sharexu@linux 1104]$ ./producer
Memory attached at 9BC49000
waiting for client...
waiting for client...
waiting for client...
waiting for client...
waiting for client...
Enter some text: aaaa
waiting for client...
Enter some text: bbbbbb
waiting for client...
waiting for client...
Enter some text: ccc
waiting for client...
waiting for client...
Enter some text: end
[sharexu@linux 1104]$

```

图 11-6 例 11.4 生产者执行结果图

```

[sharexu@linux 1104]$ ./consumer
Memory attached at 3504A000
You wrote: aaaa
You wrote: bbbbbb
You wrote: ccc
You wrote: end

```

图 11-7 例 11.4 消费者执行结果图

程序大体思路为：程序 consumer 创建共享内存，然后将它连接到自己的地址空间中。在共享内存的开始处使用了一个结构 struct_use_st，该结构中有个标志 written，当共享内存中有其他进程向它写入数据时，共享内存中的 written 被设置为 0，程序等待；当它不为 0 时，表示没有进程向共享内存写入数据，程序就从共享内存中读取数据并输出，然后重新设置共享内存中 written 的值为 0，即让其可被 shmwrite 进程写入数据。程序 producer 取得共享内存并连接到自己的地址空间中，检查共享内存中的 written 是否为 0：若不是 0，表示共享内存中的数据还没有被完，则等待其他进程读取完成，并提示用户等待；若是 0，表示没有其他进程对共享内存进行读取，则提示用户输入文本，并再次设置共享内存中的 written 为 1，表示写完成，其他进程可对共享内存进行读操作。

再来看下详细的代码讲解。

consumer.cpp 中，用 shmget 创建了一个 shared_use_st 结构体大小的共享内存，代码如下：

```

int shmid;
srand((unsigned int) getpid());
shmid=shmget((key_t)1234,sizeof(struct shared_use_st),0666|IPC_CREAT);
if (shmid == -1) {
    fprintf(stderr, "shmget failed\n");
    exit(EXIT_FAILURE);
}

```

将该共享内存映射到进程的地址空间上，代码如下：

```

void *shared_memory = (void *)0;
shared_memory = shmat(shmid, (void *)0, 0);
if (shared_memory == (void *)-1) {
    fprintf(stderr, "shmat failed\n");
    exit(EXIT_FAILURE);
}
printf("Memory attached at %X\n", (long)shared_memory);

```

将 shared_memory 分配给 shared_stuff，然后它输出 written 中的文本。循环将一直执行

到在 `written` 中找到 `end` 字符串为止。调用 `sleep` 强迫消费者程序在临界区域多待一会，让生产者程序等待，代码如下：

```
struct shared_use_st *shared_stuff;
shared_stuff = (struct shared_use_st *)shared_memory;
shared_stuff->written = 0;
int running = 1;
while(running){
    if (shared_stuff->written){
        printf("You wrote: %s", shared_stuff->text);
        sleep( rand() % 4 );
        shared_stuff->written = 0;
        if (strncmp(shared_stuff->text, "end", 3) == 0) {
            running = 0;
        }
    }
}
```

最后，共享内存被分离，然后删除，代码如下：

```
if (shmdt(shared_memory) == -1){
    fprintf(stderr, "shmdt failed\n");
    exit(EXIT_FAILURE);
}
if (shmctl(shmid, IPC_RMID, 0) == -1){
    fprintf(stderr, "shmctl(IPC_RMID) failed\n");
    exit(EXIT_FAILURE);
}
```

在 `producer.cpp` 中，前面映射共享内存的和 `consumer.cpp` 中的一样，使用相同的键值 1234 来取得并连接同一个共享内存段，然后提示用户输入一些文本。如果标志 `written` 被设置，生产者就知道消费者进程还未读完上一次的数据，因此就继续等待。当其他进程清除了这个标志后，生产者将可以写入新的数据并重新设置这个标志。它还使用字符串 `end` 来终止并分离共享内存段，代码如下：

```
while(running){
    while(shared_stuff->written == 1){
        sleep(1);
        printf("waiting for client...\n");
    }
    printf("Enter some text: ");
    fgets(buffer, BUFSIZ, stdin);
    strncpy(shared_stuff->text, buffer, TEXT_SZ);
    shared_stuff->written = 1;
    if (strncmp(buffer, "end", 3) == 0) {
        running = 0;
    }
}
```

事实上，这个程序是不安全的，当有多个程序同时向共享内存中读写数据时，问题就会出现。可能你会认为，可以改变一下 `written` 的使用方式来解决这个问题。例如，只有当 `written` 为 0 时进程才可以向共享内存写入数据，而只有当 `written` 不为 0 时才能对其进行读取，同时把 `written` 进行加 1 操作，读取完后进行减 1 操作，这就有点像文件锁中的读写锁的功能。

但实际上这都不是原子操作，所以这种做法是不可行的。试想当 `written` 为 0 时，如果有两个进程同时访问共享内存，它们就会发现 `written` 为 0，于是两个进程都对其进行写操作，这种操作显然不行。当 `written` 的值为 1 时，有两个进程同时对共享内存进行读操作时也是如此，当这两个进程都读取完时，`written` 的值就变成了 -1。

要想让程序安全地执行，就要有一种进程同步的进制，保证在进入临界区的操作是原子操作。例如，可以使用前面所讲的信号量来进行进程的同步。因为对信号量的操作都是原子性的。

使用共享内存的优缺点如下所述。

(1) 优点：使用共享内存进行进程间的通信非常方便，而且函数的接口也简单，数据的共享还使进程间的数据不用传送，而是直接访问内存，也加快了程序的效率。同时，它也不像无名管道那样要求通信的进程有一定的父子关系。

(2) 缺点：共享内存没有提供同步的机制，这使得在使用共享内存进行进程间通信时，往往要借助其他的手段来进行进程间的同步工作。

11.4 信号量

第 9 章中用于多线程同步的方式中已经提及信号量，但用于多线程同步的信号量是 POSIX 信号量，而本节即将要展开的是 SYSTEM V 信号量，本质上说这两种都是用户态进程可以使用的信号量。SYSTEM V 信号量，下面简称为信号量。

在 Linux 中提供了一组函数接口用于使用信号量，首先常用的函数是 `semget`，该函数用来创建和打开信号量，它用到的头文件是：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

函数原型是：

```
int semget(key_t key, int nsems, int semflg);
```

该函数执行成功返回信号量标示符，失败则返回 -1。参数 `key` 是函数通过调用 `ftok` 函数得到的键值，`nsems` 代表创建信号量的个数，如果只是访问而不创建则可以指定该参数为 0；但一旦创建了该信号量，就不能更改其信号量个数。只要不删除该信号量，就可以重新调用该函数创建该键值的信号量，该函数只是返回以前创建的值，而不会重新创建。`semflg`

指定该信号量的读写权限，当创建信号量时不许加 `IPC_CREAT`，若指定 `IPC_CREAT | IPC_EXCL` 后创建时发现存在该信号量，创建失败。

`semop` 函数，用于改变信号量的值，原型是：

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

`sem_id` 是由 `semget` 返回的信号量标识符，`sembuf` 结构的定义如下：

```
struct sembuf{
    short sem_num;        // 除非使用一组信号量，否则它为 0
    short sem_op;         // 信号量在一次操作中需要改变的数据，通常是两个数，
                          // 一个是 -1，即 P（等待）操作，一个是 +1，即 V（发送信号）操作。
    short sem_flg;        // 通常为 SEM_UNDO，使操作系统跟踪信号，
                          // 并在进程没有释放该信号量而终止时，操作系统释放信号量
};
```

`semctl` 函数，该函数用来直接控制信号量信息，它的原型是：

```
int semctl(int semid, int semnum, int cmd, ...);
```

如果有第 4 个参数，它通常是一个 `union semun` 结构，定义如下：

```
union semun{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
```

前两个参数与前面一个函数中的一样，`cmd` 通常是 `SETVAL` 或 `IPC_RMID`。`SETVAL` 用来把信号量初始化为一个已知的值。`p` 值通过 `union semun` 中的 `val` 成员设置，其作用是在信号量第一次使用前对它进行设置。`IPC_RMID` 用于删除一个已经无须继续使用的信号量标识符。

共享内存是进程间通信的最快的方式，但是共享内存的同步问题自身无法解决（即进程该何时去共享内存取得数据，而何时不能取），但用信号量即可轻易解决这个问题。下面使用例 11.5 来说明如何使用信号量解决共享内存的同步问题。这个例子的主要功能是 `writer` 向 `reader` 传递数据，并且只有在 `writer` 发送完毕后，`reader` 才取数据，否则阻塞等待。

【例 11.5】 使用信号量实现进程间通信。

`reader.cpp` 的代码是：

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```

#include <string.h>
#include <sys/shm.h>
#include <errno.h>
#define SEM_KEY 4001
#define SHM_KEY 5678
union semun {
    int val;
};
int main(void){
    /*create a shm*/
    int semid,shmidx;
    shmidx = shmget(SHM_KEY,sizeof(int),IPC_CREAT|0666);
    if(shmidx<0){
        printf("create shm error\n");
        return -1;
    }
    void * shmidxptr;
    shmidxptr =shmat(shmidx,NULL,0);
    if(shmidxptr == (void *)-1){
        printf("shmat error:%s\n",strerror(errno));
        return -1;
    }
    int * data = (int *)shmidxptr;
    semid = semget(SEM_KEY,2,IPC_CREAT|0666);/* 这里是创建一个 semid, 并且有两个信号量 */
    union semun semun1;/* 下面这四行就是初始化那两个信号量, 一个 val=0, 另一个 val=1 */
    semun1.val=0;
    semctl(semid,0,SETVAL,semun1);
    semun1.val=1;
    semctl(semid,1,SETVAL,semun1);
    struct sembuf sembuf1;
    while(1){
        sembuf1.sem_num=0;/*sem_num=0 指的是下面操作指向第一个信号量, 上面设置可知其 val=0*/
        sembuf1.sem_op=-1; /* 初始化值为 0, 再 -1 的话就会等待 */
        sembuf1.sem_flg=SEM_UNDO;
        semop(semid,&sembuf1,1);/*reader 在这里会阻塞, 直到收到信号 */
        printf("the NUM:%d\n",*data);/* 输出结果 */
        sembuf1.sem_num=1;/* 这里让 writer 再次就绪, 就这样循环 */
        sembuf1.sem_op=1;
        sembuf1.sem_flg=SEM_UNDO;
        semop(semid,&sembuf1,1);
    }
    return 0;
}

```

writer.cpp 的代码是:

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <unistd.h>

```

```

#include <stdlib.h>
#include <string.h>
#include <sys/shm.h>
#include <errno.h>
#define SEM_KEY 4001
#define SHM_KEY 5678
union semun {
    int val;
};
int main(void) {
    /*create a shm*/
    int semid,shmld;
    shmld = shmget(SHM_KEY,sizeof(int),IPC_CREAT|0666);
    if(shmld<0){
        printf("create shm error\n");
        return -1;
    }
    void * shmpttr;
    shmpttr =shmat(shmld,NULL,0);
    if(shmpttr == (void *)-1){
        printf("shmat error:%s\n",strerror(errno));
        return -1;
    }
    int * data = (int *)shmpttr;
    semid = semget(SEM_KEY,2,0666);
    struct sembuf sembuf1;
    union semun semun1;
    while(1){
        sembuf1.sem_num=1; /* 这里指向第 2 个信号量 (sem_num=1) */
        sembuf1.sem_op=-1; /* 操作是 -1, 因为第 2 个信号量初始值为 1, 所以下面不会阻塞 */
        sembuf1.sem_flg=SEM_UNDO;
        semop(semid,&sembuf1,1); /* 继续 */
        scanf("%d",data); /* 用户在终端输入数据 */
        sembuf1.sem_num=0; /* 这里指向第一个信号量 */
        sembuf1.sem_op=1; /* 操作加 1 */
        sembuf1.sem_flg=SEM_UNDO;
        semop(semid,&sembuf1,1);
        /* 执行加 1 后, 我们发现, reader 阻塞正是由于第一个信号量为 0, 无法减 1, 而现在 writer 先为其加 1,
        那 reader 就绪后 writer 继续循环, 发现第二个信号量已经减为 0, 则阻塞了, 我们回到 reader */
    }
    return 0;
}

```

程序的执行结果如图 11-8 所示。

reader.cpp 中, 先获得共享内存, 并把进程连接到该地址空间上, 代码如下:

图 11-8 例 11.5 程序执行结果图

```

shmld = shmget(SHM_KEY,sizeof(int),IPC_CREAT|0666);
if(shmld<0){
    printf("create shm error\n");
}

```

```

    return -1;
}
void * shmptr;
shmptr = shmat(shmid, NULL, 0);
if(shmptr == (void *)-1){
    printf("shmat error:%s\n", strerror(errno));
    return -1;
}
int * data = (int *)shmptr;

```

创建并初始化 2 个信号量，代码如下所示。第一个初始值为 0，第二个初始值为 1。

```

semid = semget(SEM_KEY, 2, IPC_CREAT|0666); /* 这里是创建一个 semid，并且有两个信号量 */
union semun semun1; /* 下面这四行就是初始化那两个信号量，一个 val=0，另一个 val=1 */
semun1.val=0;
semctl(semid, 0, SETVAL, semun1);
semun1.val=1;
semctl(semid, 1, SETVAL, semun1);
struct sembuf sembuf1;

```

reader 要等到 writer 有数据之后，才开始读；把数据读到之后，让 writer 可以继续写，代码如下：

```

while(1){
    sembuf1.sem_num=0; /* sem_num=0 指的是下面操作指向第一个信号量，上面设置可知其 val=0 */
    sembuf1.sem_op=-1; /* 初始化为 0，再 -1 的话就会等待 */
    sembuf1.sem_flg=SEM_UNDO;
    semop(semid, &sembuf1, 1); /* reader 在这里会阻塞，直到收到信号 */
    printf("the NUM:%d\n", *data); /* 输出结果 */
    sembuf1.sem_num=1; /* 这里让 writer 再次就绪，就这样循环 */
    sembuf1.sem_op=1;
    sembuf1.sem_flg=SEM_UNDO;
    semop(semid, &sembuf1, 1);
}

```

writer.cpp 连接到同一个共享内存，并接收相同的信号量，代码如下：

```

shmid = shmget(SHM_KEY, sizeof(int), IPC_CREAT|0666);
if(shmid<0){
    printf("create shm error\n");
    return -1;
}
void * shmptr;
shmptr = shmat(shmid, NULL, 0);
if(shmptr == (void *)-1){
    printf("shmat error:%s\n", strerror(errno));
    return -1;
}
int * data = (int *)shmptr;
semid = semget(SEM_KEY, 2, 0666);

```

先写数据，写完后改变信号，让读者可以读，代码如下：

```
while(1){
    sembuf1.sem_num=1; /* 这里指向第 2 个信号量 (sem_num=1) */
    sembuf1.sem_op=-1; /* 操作是 -1，因为第 2 个信号量初始值为 1，所以下面不会阻塞 */
    sembuf1.sem_flg=SEM_UNDO;
    semop(semid,&sembuf1,1); /* 继续 */
    scanf("%d",&data); /* 用户在终端输入数据 */
    sembuf1.sem_num=0; /* 这里指向第一个信号量 */
    sembuf1.sem_op=1; /* 操作加 1 */
    sembuf1.sem_flg=SEM_UNDO;
    semop(semid,&sembuf1,1);
    /* 执行 +1 后，我们发现，reader 阻塞正是由于第一个信号量为 0，无法减一，而现在 writer 先为其加 1，
       那 reader 就绪！writer 继续循环，发现第二个信号量已经减为 0，则阻塞了，我们回到 reader */
}
```

多打开几个终端，同时执行 writer 程序，看是否 reader 能够正确地读到数据，即是否能得到如图 11-9 所示的结果。

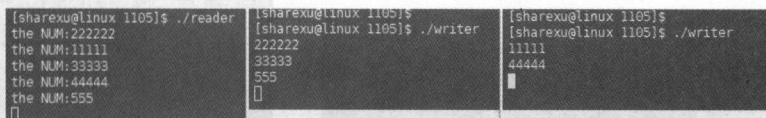


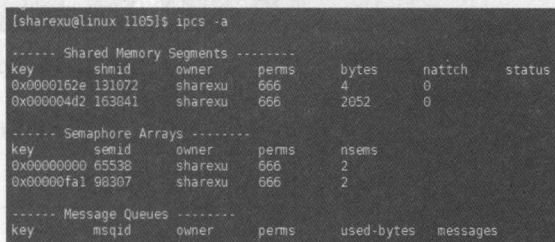
图 11-9 例 11.5 程序多个终端同时执行 write 结果图

程序结果显示，reader 程序可以读到多个 writer 发送的数据。于是，就实现了用信号量共享内存的同步。

11.5 ipcs 命令

ipcs 是一个 UNIX/Linux 的命令，用于报告系统的消息队列、信号量、共享内存等。下面列举一些常用命令。

(1) ipcs -a 用于列出本用户所有相关的 ipcs 参数，结果如图 11-10 所示。



```
[sharexu@linux 1105]$ ipcs -a
----- Shared Memory Segments -----
key      shmid   owner    perms    bytes    nattch   status
0x0000162e 131072  sharexu  666      4         0
0x000004d2 163041  sharexu  666     2052      0

----- Semaphore Arrays -----
key      semid    owner    perms    nsems
0x00000000 65538   sharexu  666      2
0x00000fa1 98307   sharexu  666      2

----- Message Queues -----
key      msgqid   owner    perms    used-bytes  messages
```

图 11-10 ipcs -a 命令执行结果图

(2) ipcs -q 用于列出进程中的消息队列，结果如图 11-11 所示。

(3) `ipcs -s` 用于列出所有的信号量，结果如图 11-12 所示。

```
[sharexu@linux 1103]$ ipcs -q
----- Message Queues -----
key      msgqid  owner   perms   used-bytes   messages
0x000004d2 98304   sharexu 666     0             0
```

图 11-11 `ipcs -q` 命令执行结果图

```
[sharexu@linux 1105]$ ipcs -s
----- Semaphore Arrays -----
key      semid   owner   perms   nsems
0x00000000 65538   sharexu 666     2
0x00000fa1 98307   sharexu 666     2
```

图 11-12 `ipcs -s` 命令执行结果图

(4) `ipcs -m` 用于列出所有的共享内存信息，结果如图 11-13 所示。

(5) `ipcs -l` 用于列出系统的限额，结果如图 11-14 所示。

```
[sharexu@linux 1105]$ ipcs -m
----- Shared Memory Segments -----
key      shmid   owner   perms   bytes   nattch   status
0x0000162e 131072   sharexu 666     4       1        0
0x000004d2 163841   sharexu 666     2052    0        0
```

图 11-13 `ipcs -m` 命令执行结果图

```
[sharexu@linux 1103]$ ipcs -l
----- Shared Memory Limits -----
max number of segments = 4096
max seg size (kbytes) = 2097152
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1

----- Semaphore Limits -----
max number of arrays = 128
max semaphores per array = 5010
max semaphores system wide = 641280
max ops per semop call = 5010
semaphore max value = 32767

----- Messages: Limits -----
max queues system wide = 972
max size of message (bytes) = 65536
default max size of queue (bytes) = 65536
```

图 11-14 `ipcs -l` 命令执行结果图

(6) `ipcs -t` 用于列出最后的访问时间，结果如图 11-15 所示。

(7) `ipcs -u` 用于列出当前的使用情况，结果如图 11-16 所示。

```
[sharexu@linux 1103]$ ipcs -t
----- Shared Memory Attach/Detach/Change Times -----
shmid   owner   attached      detached      changed
131072   sharexu Jan  2 22:11:53 Jan  2 22:12:12 Jan  2 21:22:57
163841   sharexu Jan  2 21:32:46 Jan  2 21:34:21 Jan  2 21:32:38

----- Semaphore Operation/Change Times -----
semid   owner   last-op      last-changed
65538   sharexu Sat Jan  2 20:55:22 2016 Sat Jan  2 20:55:22 2016
98307   sharexu Sat Jan  2 22:12:12 2016 Sat Jan  2 22:11:53 2016

----- Message Queues Send/Recv/Change Times -----
msgqid  owner   send      recv      change
98304    sharexu Not set    Not set    Jan  2 22:12:32
```

图 11-15 `ipcs -t` 命令执行结果图

```
[sharexu@linux 1103]$ ipcs -u
----- Shared Memory Status -----
segments allocated 2
pages allocated 2
pages resident 2
pages swapped 0
Swap performance: 0 attempts 0 successes

----- Semaphore Status -----
used arrays = 4
allocated semaphores = 6

----- Messages: Status -----
allocated queues = 1
used headers = 0
used space = 0 bytes
```

图 11-16 `ipcs -u` 命令执行结果图

11.6 本章小结

本章主要介绍了管道、消息队列、共享内存和信号量的使用方法，它们各有各的特点及优势，应按照实际情况，选择合适的方式应用到程序中。

前面几章都是长连接的内容，接下来的第 12 章将会介绍短连接的知识。

HTTP 协议

协议是指计算机通信网络中两台计算机之间进行通信所必须共同遵守的规定或规则。HTTP(Hypertext Transfer Protocol, 超文本传输协议)是一种详细规定了浏览器和万维网(World Wide Web, WWW)服务器之间互相通信的规则,通过因特网传送万维网文档的数据传送协议。HTTP 协议可以使浏览器更加高效地运行,使网络传输效率更高。它不仅保证计算机正确快速地传输超文本文档,还确定传输文档中的哪一部分内容首先显示(如文本先于图形)等。

HTTP 由于其灵活、简单、快速的特点,应用非常广泛。浏览网页是 HTTP 的主要应用,但是这并不代表 HTTP 就只能应用于网页的浏览。HTTP 是一种协议,只要通信的双方都遵守这个协议,HTTP 就能有用武之地,比如常用的 QQ、迅雷这些软件,都使用了 HTTP 协议。

12.1 HTTP 协议工作流程

在网络七层模型中,HTTP 是在应用层,也就是在传输层以上。事实上,HTTP 是基于 TCP 协议的;而我们常说的 HTTPS 协议,则是同处应用层而基于 TLS、SSL 协议层之上的协议,两者的区别如图 12-1 所示。

HTTP (应用层)	HTTPS (应用层)
TCP (传输层)	TLS SSL (应用层)
IP (网络层)	TCP (传输层)
数据链路层 (又称网络接口层)	IP (网络层)
	数据链路层 (又称网络接口层)

图 12-1 HTTP 协议与 HTTPS 协议的对比

HTTP 默认的端口号为 80，HTTPS 的默认端口号则为 443。

HTTP 是基于传输层的 TCP 协议，而 TCP 是一个端到端的面向连接的协议。所谓的“端到端”可以理解为进程到进程之间的通信，所以 HTTP 在开始传输之前，首先需要建立 TCP 连接，而 TCP 连接的过程需要进行“三次握手”，在 TCP 三次握手之后，建立了 TCP 连接，此时 HTTP 就可以进行传输了。在 HTTP1.1 中（通过 Connection 头设置）默认在 HTTP 传输完成后不断开 TCP 连接。在此之前的版本 HTTP 则默认是断开连接，也就是同一个客户端的这次请求和上次请求是没有对应关系的。

一次 HTTP 操作称为一个事务，其工作过程可分为以下 4 步。

- (1) 首先客户机与服务器需要建立连接。只要单击某个超级链接，HTTP 的工作即开始。
- (2) 建立连接后，客户机发送一个请求给服务器，请求方式的格式为：统一资源标识符 (URL)、协议版本号，后边是 MIME 信息（包括请求修饰符、客户机信息和可能的内容）。
- (3) 服务器接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、一个成功或错误的代码，后边是 MIME 信息（包括服务器信息、实体信息和可能的内容）。
- (4) 客户端接收服务器所返回的信息通过浏览器显示在用户的显示屏上，然后客户机与服务器断开连接。

如果在以上过程中的某一步出现错误，那么产生错误的信息将返回到客户端，由显示屏输出。对于用户来说，这些过程是由 HTTP 自己完成的，用户只要用鼠标操作，等待信息显示就可以了。

HTTP 协议永远都是客户端发起请求，服务器回送响应，这样就会使得无法实现客户端未发起的请求，而服务器将消息推送给客户端。

12.2 HTTP 协议结构

HTTP 协议无论是请求报文还是回应报文，都分为以下 4 个部分。

(1) 报文头 (initial line)，上面的例子中的“GET http://www.baidu.com/favicon.ico HTTP/1.1”表示用 GET 方法请求 http://www.baidu.com/favicon.ico 这个文件，用的是 HTTP/1.1 协议。

(2) 0 个或多个请求头 (header line)，例如 Accept-Language: en。

(3) 空行（作为 header lines 的结束）。

(4) 可选的消息体。

HTTP 协议是基于行的协议，每一行以 \r\n 作为分隔符。报文头通常表明报文的类型（例如请求类型），且报文头只占一行；请求头附带一些特殊信息，每一个请求头占一行，其格式为 name:value，即以分号作为分隔；空行也就以一个 \r\n 分隔；可选 body 通常包含数据，例如服务器返回的某个静态 HTML 文件的内容。

wireshark 是一个抓包的好工具，它能够记录计算机和互联网之间的通信内容。此处将用 wireshark 抓一个包来进行分析。

先打开 wireshark，并开启过滤器，只抓 80 端口上的包，如图 12-2 所示。



图 12-2 在 wireshark 的过滤器窗口中输入 tcp.port=80

在浏览器的地址栏中输入“http://www.baidu.com/favicon.ico”，会看到 wireshark 中抓到了这个包，如图 12-3 所示。

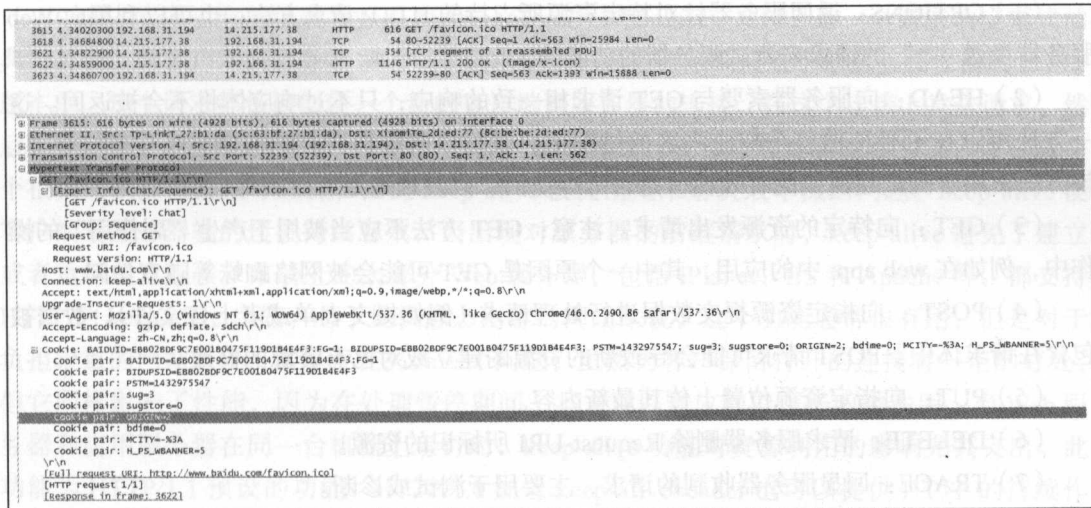


图 12-3 访问百度 icon 文件时用 wireshark 抓包结果图

图 12-3 中看不清请求包的具体内容，请求包的具体内容如图 12-4 所示。

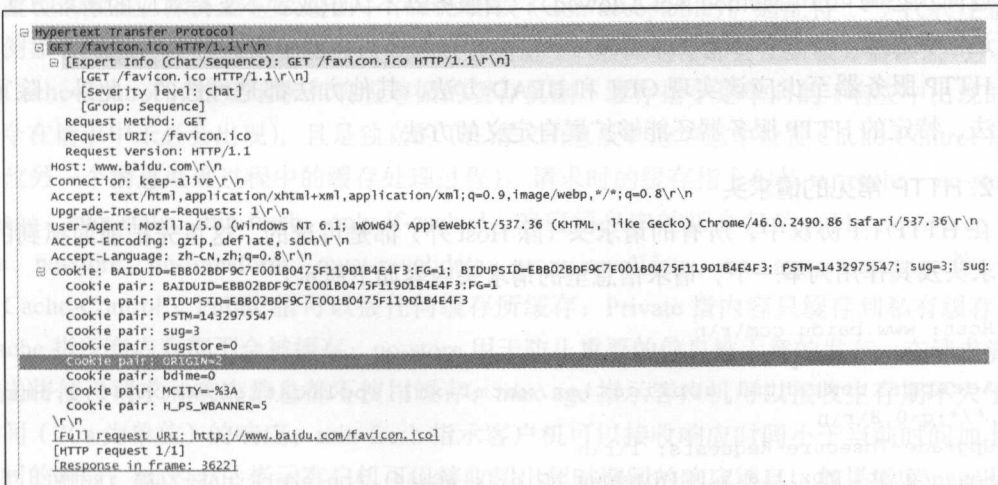


图 12-4 访问百度 icon 文件请求包内容

可以看到第一行是：

```
GET /favicon.ico HTTP/1.1\r\n
```

这个表明用 GET 方法，HTTP/1.1 协议获取 favicon.ico 文件。

1. HTTP 请求方法

HTTP/1.1 协议中共定义了 9 种方法（有时也叫“动作”）来表明 Request-URI 指定的资源的不同操作方式，如下所述。

(1) OPTIONS：返回服务器针对特定资源所支持的 HTTP 请求方法；也可以利用向 Web 服务器发送“*”的请求来测试服务器的功能性。

(2) HEAD：向服务器索要与 GET 请求相一致的响应，只不过响应体将不会被返回。这一方法可以在不必传输整个响应内容的情况下，就可以获取包含在响应消息头中的元信息。该方法常用于测试超链接的有效性，是否可以访问，以及最近是否更新等信息。

(3) GET：向特定的资源发出请求。注意：GET 方法不应当被用于产生“副作用”的操作中，例如在 web app. 中的应用，其中一个原因是 GET 可能会被网络蜘蛛等随意访问。

(4) POST：向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立或对已有资源的修改。

(5) PUT：向指定资源位置上传其最新内容。

(6) DELETE：请求服务器删除 Request-URI 所标识的资源。

(7) TRACE：回显服务器收到的请求，主要用于测试或诊断。

(8) CONNECT：HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。

(9) PATCH：用来将局部修改应用于某一资源，该操作添加于规范 RFC5789 中。

方法名称是区分大小写的。当某个请求所针对的资源不支持对应的请求方法时，服务器应当返回状态码 405 (Method Not Allowed)；当服务器不认识或者不支持对应的请求方法时，应当返回状态码 501 (Not Implemented)。

HTTP 服务器至少应该实现 GET 和 HEAD 方法，其他方法都是可选的。此外，除了上述方法，特定的 HTTP 服务器还能够扩展自定义的方法。

2. HTTP 常见的请求头

在 HTTP/1.1 协议中，所有的请求头（除 Host 外）都是可选的。这里先把前面抓到的包的请求头及其作用列举一下，请求信息里的请求头内容如下所示：

```
Host: www.baidu.com\r\n
Connection: keep-alive\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8\r\n
Upgrade-Insecure-Requests: 1\r\n
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111 Safari/537.36\r\n
```

```
Accept-Encoding: gzip, deflate, sdch\r\n
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6\r\n
```

Host : (发送请求时, 该请求头是必需的) 主要用于指定被请求资源的 Internet 主机和端口号, 它通常从 HTTP URL 中提取出来的。HTTP/1.1 请求必须包含主机头域, 否则系统会以 400 状态码返回。例如上面的例子中, 在浏览器中输入 “http://www.baidu.com/favicon.ico” 后, 浏览器发送的请求消息中, 就会包含 Host 请求头域 “Host : http://www.baidu.com”, 此处使用默认端口号 80, 若指定了端口号, 则变成: “Host: 指定端口号”。

Connection : 它的值通常只有两个, keep-alive 和 close。HTTP 是一个请求响应模式的典型范例, 即客户端向服务器发送一个请求信息, 服务器来响应这个信息。在之前的 HTTP 版本中, 每个请求都将被创建一个新的客户端到服务器的连接, 在这个连接上发送请求, 然后接收请求。这样的模式有一个很大的优点就是简单, 很容易理解和编程实现; 但也有一个很大的缺点就是效率很低, 因此 keep-alive 被提出用来解决效率低的问题。keep-alive 使客户端到服务器端的连接持续有效, 当出现对服务器的后继请求时, keep-alive 避免了建立或者重新建立连接。市场上的大部分 Web 服务器, 包括 iPlanet、IIS 和 Apache 等, 都支持 HTTP 的 keep-alive 功能。对于提供静态内容的网站来说, 这个功能通常很有用; 但是对于负担较重的网站来说, 这里存在另外一个问题: 虽然为客户保留打开的连接有一定的好处, 但它同样影响了性能, 因为在处理暂停期间, 本来可以释放的资源仍旧被占用。当 Web 服务器和应用服务器在同一台机器上运行时, keep-alive 功能对资源利用的影响尤其突出。此功能为 HTTP/1.1 预设的功能, HTTP/1.0 加上 keep-aliveheader 也可以提供 HTTP 的持续作用功能。

Accept : 浏览器端可以接收的 MIME 类型。例如: Accept: text/html 代表浏览器可以接收服务器回发的类型为 text/html, 也就是我们常说的 html 文档, 如果服务器无法返回 text/html 类型的数据, 服务器应该返回一个 406 错误 (non acceptable)。通配符 (*) 代表任意类型, 例如 Accept: */* 代表浏览器可以处理所有类型 (一般浏览器都会发给服务器该语句)。

Cache-Control : 指定请求和响应遵循的缓存机制。缓存指令是单向的 (响应中出现的缓存指令在请求中未必会出现), 且是独立的 (在请求消息或响应消息中设置 Cache-Control 并不会修改另一个消息处理过程中的缓存处理过程)。请求时的缓存指令包括 no-cache、no-store、max-age、max-stale、min-fresh、only-if-cached, 响应消息中的指令包括 public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age、s-maxage 等。

Cache-Control : Public 指可以被任何缓存所缓存; Private 指内容只缓存到私有缓存中; no-cache 指所有内容都不会被缓存; no-store 用于防止重要的信息被无意的发布, 在请求消息中发送将使得请求和响应消息都不使用缓存; max-age 指示客户机可以接收生存期不大于指定时间 (以 s 为单位) 的响应; min-fresh 指示客户机可以接收响应时间小于当前时间加上指定时间的响应; max-stale 指示客户机可以接收超出超时期间的响应消息, 如果指定 max-stale 消息的值, 那么客户机可以接收超出超时期指定值之内的响应消息。

Accept-Encoding：浏览器声明自己可接收的编码方法，通常说明是否支持压缩并指定压缩方法；Servlet 能够向支持 gzip 的浏览器返回经 gzip 编码的 HTML 页面。许多情形下这可以减少 5 ~ 10 倍的下载时间。例如：Accept-Encoding: gzip, deflate。如果请求消息中没有设置这个域，服务器假定客户端对各种内容编码都可以接受。

Accept-Language：浏览器声明自己接收的语言。语言跟字符集的区别可以理解为：中文是语言，中文有多种字符集，比如 big5、gb2312、gbk 等；例如：Accept-Language: en-us。如果请求消息中没有设置这个报头域，则服务器假定客户端对各种语言都可以接受。

Accept-Charset：浏览器可接受的字符集。如果在请求消息中没有设置这个域，默认时表示任何字符集都可以接受。

User-Agent：用于告诉 HTTP 服务器，客户端使用的操作系统和浏览器的名称和版本。

上面已经访问了一次 <http://www.baidu.com/favicon.ico>，在不清除浏览器缓存的情况下，如果再访问一次，可以看到返回码是 304，如图 12-5 所示。

3148	3.80269600	192.168.31.194	14.215.177.38	HTTP	729 GET /favicon.ico HTTP/1.1
3149	3.81409800	14.215.177.38	192.168.31.194	TCP	66 80-53441 [SYN, ACK] Seq=0 Ack=1 W
3150	3.81414100	192.168.31.194	14.215.177.38	TCP	54 53441-80 [ACK] Seq=1 Ack=1 win=17
3151	3.81522900	14.215.177.38	192.168.31.194	TCP	54 80-53427 [ACK] Seq=1 Ack=676 win=
3152	3.81588300	14.215.177.38	192.168.31.194	HTTP	222 HTTP/1.1 304 Not Modified

图 12-5 非首次访问一个未修改的资源文件的返回值是 304

也可以来对比下这前后两个请求包中的请求信息，第一个如图 12-4 所示，第二个如图 12-6 所示。

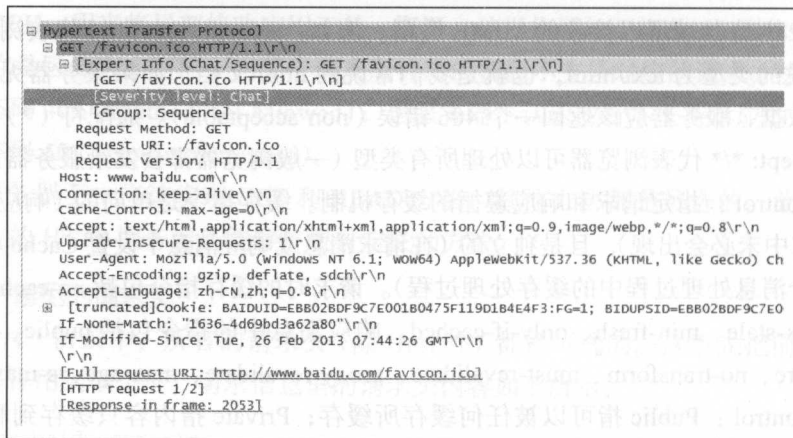


图 12-6 非首次访问资源文件请求包内容图

可见，第二个请求包中的内容，比第一个多了以下两行：

```
If-None-Match: "1636-4d69bd3a62a80"\r\n
If-Modified-Since: Tue, 26 Feb 2013 07:44:26 GMT\r\n
```


If-Modified-Since: 把浏览器端缓存页面的最后修改时间发送到服务器去, 服务器会把这个时间与服务器上实际文件的最后修改时间进行对比。如果时间一致, 那么返回 304 状态, 客户端就直接使用本地缓存文件; 如果时间不一致, 就会返回 200 状态和新的文件内容。客户端接到之后, 会丢弃旧文件, 把新文件缓存起来, 并显示在浏览器中。

If-None-Match: If-None-Match 和 ETag 一起工作, 工作原理是在 HTTP Response 中添加 ETag 信息。当用户再次请求该资源时, 将在 HTTP Request 中加入 If-None-Match 信息 (ETag 的值)。如果服务器验证资源的 ETag 没有改变 (该资源没有更新), 将返回一个 304 状态告诉客户端使用本地缓存文件。否则将返回 200 状态和新的资源和 Etag。使用这样的机制可以提高网站的性能。

我们再来看下其他常见的请求头:

Pragma: 指定 no-cache 值表示服务器必须返回一个刷新后的文档, 即使它是代理服务器而且已经有了页面的本地拷贝; 在 HTTP/1.1 版本中, 它和 Cache-Control:no-cache 作用一模一样。Pragma 只有一个用法, 例如: Pragma: no-cache。注意: 在 HTTP/1.0 版本中, 只实现了 Pragma:no-cache 功能, 没有实现 Cache-Control 功能。

Content-Type, 例如: Content-Type: application/x-www-form-urlencoded。

Referer: 包含一个 URL, 用户从该 URL 代表的页面出发访问当前请求的页面。提供了 Request 的上下文信息的服务器, 告诉服务器其是从哪个链接转过来的。比如从个人主页上链接到一个朋友那里, 对方的服务器就能够从 HTTP Referer 中统计出每天有多少用户通过单击我主页上的链接访问他的网站。例如: Referer:http://translate.google.cn/?hl=zh-cn&tab=wT。

Cookie: 最重要的请求头之一, 用于将 cookie 的值发送给 HTTP 服务器。

Content-Length: 表示请求消息正文的长度。例如: Content-Length: 38。

Authorization: 授权信息, 通常出现在对服务器发送的 WWW-Authenticate 头的应答中。主要用于证明客户端有权查看某个资源。当浏览器访问一个页面时, 如果收到服务器的响应代码为 401(未授权), 可以发送一个包含 Authorization 请求报头域的请求, 要求服务器对其进行验证。

UA-Pixels、**UA-Color**、**UA-OS**、**UA-CPU**: 由某些版本的 IE 浏览器所发送的非标准的请求头, 表示屏幕大小、颜色深度、操作系统和 CPU 类型。

From: 请求发送者的 E-mail 地址, 由一些特殊的 Web 客户程序使用, 浏览器一般不会用到它。

Range: 可以请求实体的一个或者多个子范围。例如, 表示头 500 个字节: bytes=0-499; 表示第二个 500 字节: bytes=500-999; 表示最后 500 个字节: bytes=-500; 表示 500 字节以后的范围: bytes=500-; 第一个和最后一个字节: bytes=00/-1; 同时指定几个范围: bytes=500-600、601-999。但是服务器可以忽略此请求头, 如果无条件 GET 包含 Range 请求头, 响应会以状态码 206 (PartialContent) 返回而不是 200 (OK)。

3. HTTP 回应报文

先来看下访问 <http://www.baidu.com/favicon.ico> 得到的返回包内容, 如图 12-7 所示。

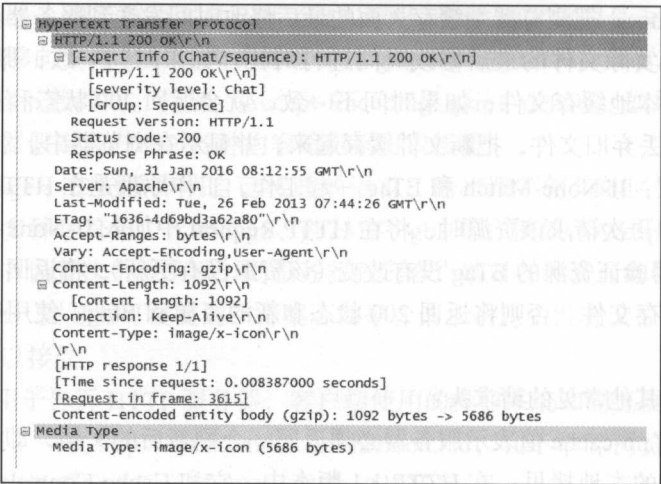


图 12-7 访问图片请求包内容图

因为访问的内容是一张图片，所以 media type 是 image。

第一行是报文头，第一个字段表明 HTTP 协议版本，可以直接以请求报文为准（即请求报文版本是多少这里就是多少）；第二个字段是一个 status code，也就是返回码，相当于请求结果，请求结果被 HTTP 官方事先定义，例如 200 表示成功、404 表示资源不存在等；最后一个字段为 status code 的可读字符串。

返回码由 3 位数字组成，第一个数字定义了响应的类别，且有 5 种可能的取值。

- (1) 1xx：指示信息，表示请求已接收，继续处理。
- (2) 2xx：成功，表示请求已被成功接收、理解、接受。
- (3) 3xx：重定向，要完成请求必须进行更进一步的操作。
- (4) 4xx：客户端错误，请求有语法错误或请求无法实现。
- (5) 5xx：服务器端错误，服务器未能实现合法的请求。

常见返回码和其状态描述说明如下：

200 OK	// 客户端请求成功
400 Bad Request	// 客户端请求有语法错误，不能被服务器所理解
401 Unauthorized	// 请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用
403 Forbidden	// 服务器收到请求，但是拒绝提供服务
404 Not Found	// 请求资源不存在，比如：输入了错误的 URL
500 Internal Server Error	// 服务器发生不可预期的错误
503 Server Unavailable	// 服务器当前不能处理客户端的请求，一段时间后可能恢复正常

Date：表示消息发送的时间，时间的描述格式由 rfc822 定义。例如，Date: Sun, 31 Jan 2016 08:12:55 GMT\r\n。Date 描述的时间表示世界标准时，换算成本地时间，需要知道用户所在的时区。

Server：指明 Web 服务器用来处理请求的软件信息。例如：Server: Microsoft-IIS/7.5、Server: Apache-Coyote/1.1。

Accept-Ranges : Web 服务器表明自己是否接收获取其某个实体的一部分（比如文件的一部分）的请求。bytes 表示接收，none 表示不接收。

Vary : Web 服务器用该头部内容告诉 Cache 服务器，在什么条件下才能用本响应所返回的对象响应后续的请求。假如源 Web 服务器在接到第一个请求消息时，其响应消息的头部为：

```
Content-Encoding: gzip; Vary: Content-Encoding
```

那么 Cache 服务器会分析后续请求消息的头部，检查其 Accept-Encoding，是否跟先前响应的 Vary 头部值一致，即是否使用相同的内容编码方法，这样就可以防止 Cache 服务器将自己 Cache 里面压缩后的实体响应传递给不具备解压能力的浏览器。例如：Vary: Accept-Encoding。

Content-Encoding : Web 服务器表明自己使用了什么压缩方法（gzip, deflate）压缩响应中的对象。

Content-Length : Web 服务器告诉浏览器自己响应的对象的长度。

Content-Type : Web 服务器告诉浏览器自己响应的对象的类型。

12.3 HTTPS

在网络上，两个实体之间的通信会面临什么样的安全问题呢？最常见的是被窃听，只要你的数据在网络上传输，就有可能被窃听，窃听者可能把窃听到的数据进行篡改、伪造，再传送给下一个人。数据的安全性成了一个永恒的课题。一句话说得很好，当你学习网络安全后，你发现网络再也不安全了。

初级的防御手段，是双方都约定一个加密的算法，把加密后的数据进行传输，收到数据方再进行解密。但更高级的方法是，公开代码、算法、协议，通过密钥的私密性来保证数据传输的安全性。这里要讲下两个概念，对称加密和非对称加密：①对称加密是指加密的密钥和解密的密钥是一样的，通常使用的有 AES 和 TEA 算法，它的特点是计算量小，又有一定的破解门槛；②而非对称加密则是指加密的密钥和解密的密钥是不一样的，也就是密钥成对出现（根据公钥无法推知私钥，根据私钥也无法推知公钥），加密解密使用不同密钥（公钥加密需要私钥解密，私钥加密需要公钥解密），它的特点是计算量大，常用的有 RSA、ECC 算法等。基于性能的考虑，一般使用非对称加密算法得出密钥，再用对称加密算法对消息内容进行加密，然后再进行传输。

HTTP 协议可以轻松抓包并获得其中的内容，是一个不安全的协议，而 HTTPS（Hypertext Transfer Protocol over Secure Socket Layer）则是以安全为目标的 HTTP 通道，简单来讲是 HTTP 的安全版。HTTPS 的安全基础是 TLS(SSL 的升级版)。它是一个 URI scheme(抽象标识符体系)，句法类同 HTTP 体系，用于安全的 HTTP 数据传输。HTTPS:URL 表明它使用了 HTTPS，但 HTTPS 的存在不同于 HTTP 的默认端口及一个加密 / 身份验证层（在 HTTP 与 TCP 之间）。这个系统的最初由网景公司进行研发，提供了身份验证与加密通信方法，现在它被广泛用于万维网上安全敏感的通信，例如交易支付方面。

1. TLS

TLS 协议可用于保护正常运行于 TCP 之上的任何应用协议的通信，如 HTTP、FTP、SMTP 或 Telnet 等高层的应用协议的通信，最常见的是用 TLS 来保护 HTTP 的通信。TLS 协议的优点在于它是与应用层协议无关的。高层的应用协议能透明地建立于 TLS 协议之上。

TLS 协议在应用层协议之前就已经完成加密算法、通信密钥的协商以及服务器的认证工作。在此之后应用层协议所传送的数据都会被加密，从而保证通信的安全性。TLS 协议使用通信双方的客户证书以及 CA 根证书，允许客户端、服务器端以一种不能被偷听的方式通信，在通信双方间建立起一条安全的、可信任的通信通道。

TLS 的工作流程如图 12-8 所示。

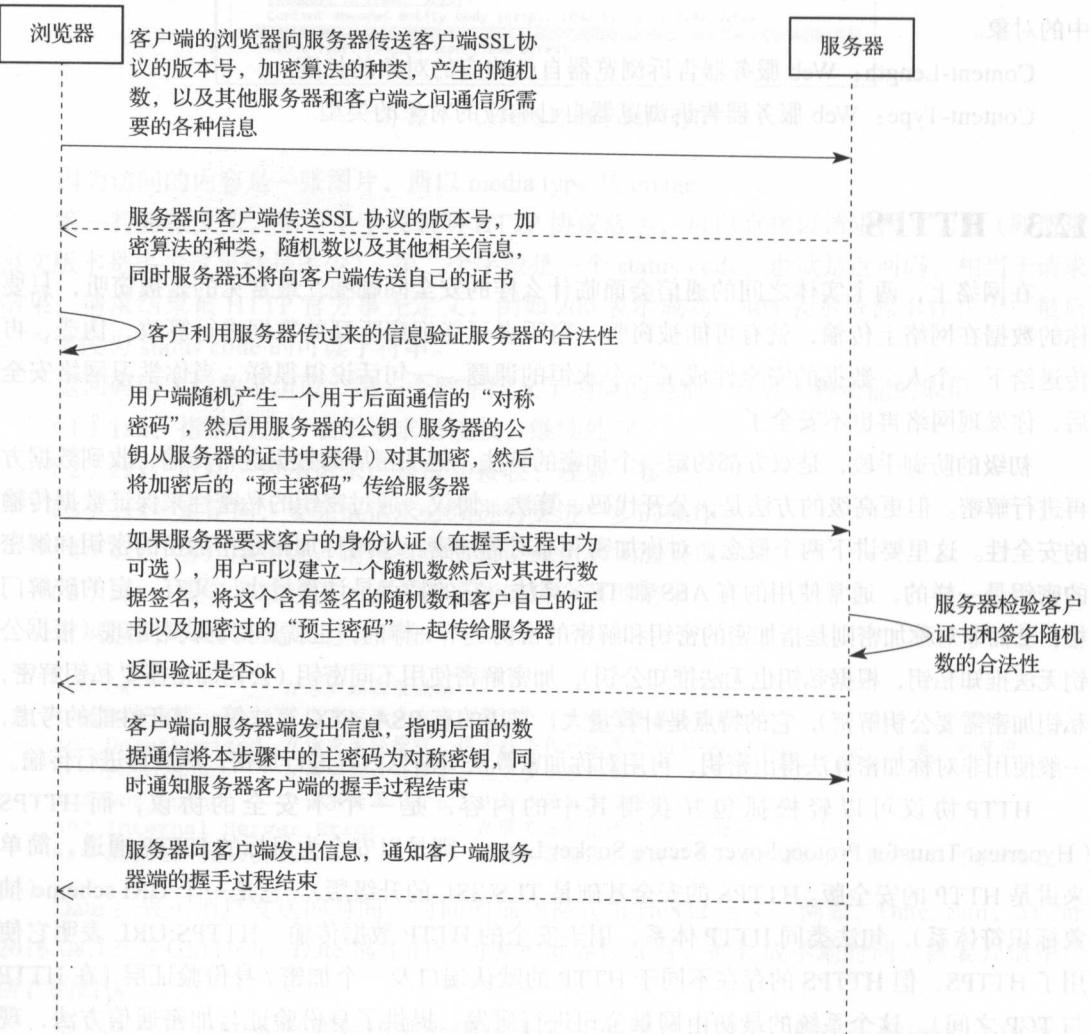


图 12-8 TLS 工作流程

TLS 协议既用到了公钥加密技术又用到了对称加密技术,对称加密技术虽然比公钥加密技术的速度快,可是公钥加密技术提供了更好的身份认证技术。TLS 的握手协议非常有效地让客户和服务端之间完成相互之间的身份认证,其主要过程如下所述。

(1) 客户端的浏览器向服务器传送客户端 TLS 协议的版本号,加密算法的种类,产生的随机数,以及其他服务器和客户端之间通信所需要的各种信息。

(2) 服务器向客户端传送 TLS 协议的版本号、加密算法的种类、随机数以及其他相关信息,同时服务器还将向客户端传送自己的证书。

(3) 客户利用服务器传过来的信息验证服务器的合法性,服务器的合法性包括:证书是否过期,发行服务器证书的 CA 是否可靠,发行者证书的公钥能否正确解开服务器证书的“发行者的数字签名”,服务器证书上的域名是否和服务器的实际域名相匹配等。如果合法性验证没有通过,通信将断开;如果合法性验证通过,将继续进行(4)。

(4) 用户端随机产生一个用于后面通信的“对称密码”,然后用服务器的公钥(由(2)步骤获取)对其加密,然后将加密后的“预主密码”传给服务器。

(5) 如果服务器要求客户的身份认证(在握手过程中为可选),用户可以产生一个随机数然后对其进行数据签名,将这个含有签名的随机数和客户自己的证书以及加密过的“预主密码”一起传给服务器。

(6) 如果服务器要求客户的身份认证,服务器必须检验客户证书和签名随机数的合法性,具体的合法性验证过程包括:①客户的证书使用日期是否有效;②为客户提供证书的 CA 是否可靠;③发行 CA 的公钥能否正确解开客户证书的发行 CA 的数字签名;④检查客户的证书是否在证书废止列表(CRL)中。检验如果没有通过,通信立刻中断;如果验证通过,服务器将用自己的私钥解开加密的“预主密码”,然后执行一系列步骤来产生主通讯密码(客户端也将通过同样的方法产生相同的主通讯密码)。

(7) 服务器和客户端用相同的主密码即“会话密码”,一个对称密钥用于 TLS 协议的安全数据通讯的加解密通信。同时在 TLS 通信过程中还要完成数据通信的完整性,防止数据通信中的任何变化。

(8) 客户端向服务器端发出消息,指明后面的数据通信将使用的(7)中的主密码为对称密钥,同时通知服务器客户端的握手过程结束。

(9) 服务器向客户端发出信息,指明后面的数据通信将使用的(7)中的主密码为对称密钥,同时通知客户端服务器端的握手过程结束。

到这里,TLS 的握手部分结束了,TLS 安全通道的数据通信开始,客户和服务端开始使用相同的对称密钥进行数据通信,同时进行通信完整性的检验。这样,基于 TLS 的 HTTPS 也就有了这样的特点:①客户端产生的密钥只有客户端和服务端能得到;②加密的数据只有客户端和服务端才能得到明文;③客户端到服务端的通信是安全的。

2. HTTPS 和 HTTP 的区别

HTTPS 和 HTTP 有以下区别。

- (1) HTTPS 协议需要到 CA 申请证书，一般免费证书很少，需要交费。
- (2) HTTP 是超文本传输协议，信息是明文传输，HTTPS 则是具有安全性的 ssl 加密传输协议。
- (3) HTTP 和 HTTPS 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。
- (4) HTTP 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 HTTP 协议安全。
- (5) 目前，HTTPS 的应用比 HTTP 的少，是因为 HTTPS 比较耗性能，对于安全性没那么高要求的应用来说，用 HTTP 就已经够了。

12.4 CGI

CGI (Common Gateway Interface, 通用网关接口) 是 HTTP 协议中最重要的技术之一，有着不可替代的重要地位。CGI 是一个 Web 服务器提供信息服务的标准接口。通过 CGI 接口，Web 服务器就能够获取客户端提交的信息，转交给服务器端的 CGI 程序进行处理，最后返回结果给客户端。组成 CGI 通信系统的是两部分：一部分是 HTML 页面，就是在用户端浏览器上显示的页面；另一部分则是运行在服务器上的 CGI 程序。它们之间的通信方式如图 12-9 所示。

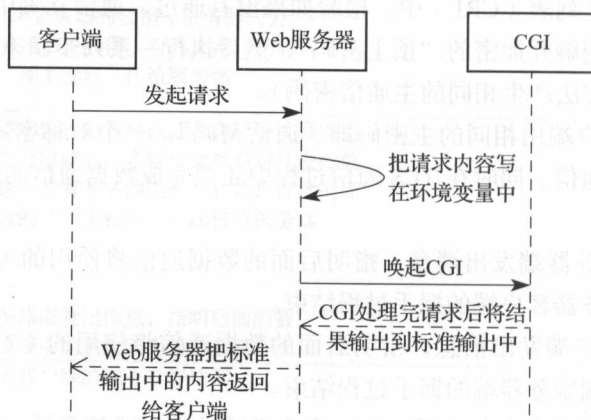


图 12-9 客户端与 CGI 的通信

服务器和客户端之间的通信，是客户端的浏览器和服务器端的 Web 服务器之间的 HTTP 通信，所以只需要知道浏览器请求执行服务器上哪个 CGI 程序就可以。一般情况下，服务器和 CGI 程序之间是通过标准输入输出来进行数据传递的，而这个过程需要环境变量的协作方可实现。环境变量在 CGI 中有着重要的地位，每个 CGI 程序只能处理一个用户请求，所以

在激活一个 CGI 程序进程时也创建了属于该进程的环境变量。后面会继续探讨环境变量，这里先不展开。

CGI 是一个标准化的协议，能够使应用程序（通常称为 CGI 程序或 CGI 脚本）同 Web 服务器和客户端进行交互。CGI 程序能够用 Python、PERL、Shell、C 或 C++ 等语言来实现。本书中只讨论使用 C++ 的相关实现。

1. Apache 安装及启动

专门处理 HTTP 请求的服务器，也被称为 Web 服务器。常用的 Web 服务器有 Apache 和 Nginx，当然几大巨头互联网公司也都有其独自研发的 Web 服务器，比如阿里巴巴的 Tengine。本书中使用 Apache 作为 Web 服务器，并按照下面 5 个步骤安装好 Apache。

(1) 下载：

```
lynx http://httpd.apache.org/download.cgi
```

(2) 解压：

```
gzip -d httpd-2.2.31.tar.gz
```

```
tar xvf httpd-2.2.31.tar
```

```
cd httpd-2.2.31
```

(3) 安装：

```
./configure --prefix=/home/sharexu/software/apache-2.2.31/apache-install
```

```
make
```

```
make install
```

(4) 将端口改为 8083：

```
vi /home/sharexu/software/apache-2.2.31/apache-install/conf/httpd.conf
```

```
Listen 8083
```

(5) 启动 Apache：

键入 “/home/sharexu/software/apache-2.2.31/apache-install/bin/apachectl -k start” 启动 apache。

(6) 在浏览器中输入 “http://42.96.142.129:8083/” 并按 Enter 键，若页面显示 “It works!” 则表示 Apache 已经安装并启动成功，如图 12-10 所示。

2. 第一个 CGI

【例 12.1】编写第一个 CGI。

```
#include <iostream>
using namespace std;
int main () {
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
```

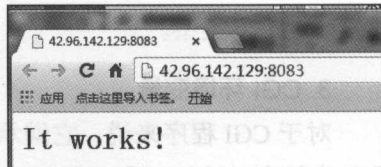


图 12-10 在浏览器中验证 Apache 是否安装成功

```
cout << "<title>Hello World - First CGI Program</title>\n";
cout << "</head>\n";
cout << "<body>\n";
cout << "<h2>Hello World! This is my first CGI program</h2>\n";
cout << "</body>\n";
cout << "</html>\n";
return 0;
}
```

编译，手动执行一次，由于本程序只是单纯输出一些信息，所以可以直接执行；之后再将其复制到 Apache 的 cgi-bin 目录下。执行结果如图 12-11 所示。

```
[sharexu@linux 1201]$ g++ -o test test.cpp
[sharexu@linux 1201]$ ./test
Content-type:text/html

<html>
<head>
<title>Hello World - First CGI Program</title>
</head>
<body>
<h2>Hello World! This is my first CGI program</h2>
</body>
</html>
[sharexu@linux 1201]$ cp test /home/sharexu/software/apache-2.2.31/apache-install/cgi-bin
```

图 12-11 例 12.1 程序的执行结果图

在浏览器中输入“http://42.96.142.129:8083/cgi-bin/test”，提示“Hello World! This is my first CGI program”，则表示运行成功了，如图 12-12 所示。

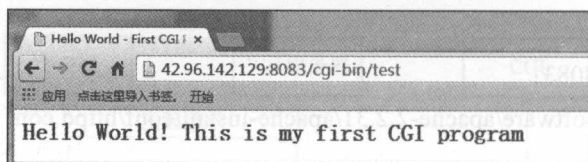


图 12-12 例 12.1 程序用浏览器访问结果图

上面的例 12.1 是一个将输出写入标准输出文件（stdout）的简单程序。这段代码中很重要的一点那就是第一行代码：Content-type:text/html\r\n\r\n，这行被发送回浏览器，指明浏览器显示的文本类型。

3. CGI 环境变量

对于 CGI 程序来说，它继承了系统的环境变量。CGI 环境变量在 CGI 程序启动时初始化，在结束时销毁。当一个 CGI 程序不是被 HTTP 服务器调用时，它的环境变量几乎是系统环境变量的复制。当这个 CGI 程序被 HTTP 服务器调用时，它的环境变量就会多了以下关于 HTTP 服务器、客户端、CGI 传输过程等项目。

CGI 相关的环境变量有 3 种：与请求相关的环境变量、与服务器相关的环境变量和与客户端相关的环境变量，详细见表 12-1。

表 12-1 CGI 相关的环境变量及其用途

与请求相关的环境变量	REQUEST_METHOD	服务器与 CGI 程序之间的信息传输方式
	QUERY_STRING	采用 GET 时所传输的信息
	CONTENT_LENGTH	STDIO 中的有效信息长度
	CONTENT_TYPE	指示所传来的信息的 MIME 类型
	CONTENT_FILE	使用 Windows HTTPd/WinCGI 标准时, 用来传送数据的文件名
	PATH_INFO	路径信息
	PATH_TRANSLATED	CGI 程序的完整路径名
	SCRIPT_NAME	所调用的 CGI 程序的名字
	HTTP_COOKIE	用户的 Cookie
与服务器相关的环境变量	GATEWAY_INTERFACE	服务器所实现的 CGI 版本
	SERVER_NAME	服务器的 IP 或名字
	SERVER_PORT	主机的端口号
	SERVER_SOFTWARE	调用 CGI 程序的 HTTP 服务器的名称和版本号
与客户端相关的环境变量	REMOTE_ADDR	客户机的主机名
	REMOTE_HOST	客户机的 IP 地址
	ACCEPT	列出能被次请求接收的应答方式
	ACCEPT_ENCODING	列出客户机支持的编码方式
	ACCEPT_LANGUAGE	表明客户机可接收语言的 ISO 代码
	AUTORIZATION	表明被证实了的用户
	FORM	列出客户机的 Email 地址
	IF_MODIFIED_SINGCE	当用 get 方式请求并且只有当文档比指定日期更早时才返回数据
	PRAGMA	设定将来要用到的服务器代理
	REFFERER	指出连接到当前文档的文档的 URL
	USER_AGENT	客户端浏览器的信息

下面来用例 12.2 看下 CGI 的环境变量的值都是怎么样的。获取环境变量需要用到 `getenv()` 函数, 它的功能是从环境中取字符串, 获取环境变量的值, 依赖的头文件为 `stdlib.h`。

【例 12.2】获取 CGI 的环境变量。

```
#include <iostream>
#include <stdlib.h>
using namespace std;
const string ENV[24] = {
    "COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
    "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
    "HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
    "HTTP_HOST", "HTTP_USER_AGENT", "PATH",
    "QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
    "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
    "SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
    "SERVER_NAME", "SERVER_PORT", "SERVER_PROTOCOL",
    "SERVER_SIGNATURE", "SERVER_SOFTWARE"};
int main(){
    cout<<"Content-type:text/html\r\n\r\n";
```

3

```
[sharexu@linux 1202]$ cp 1202 /home/sharexu/software/apache-2.2.31/apache-install/cgi-bin
```

图 12-13 例 12.2 程序执行结果图

值,如图 12-14 所示。

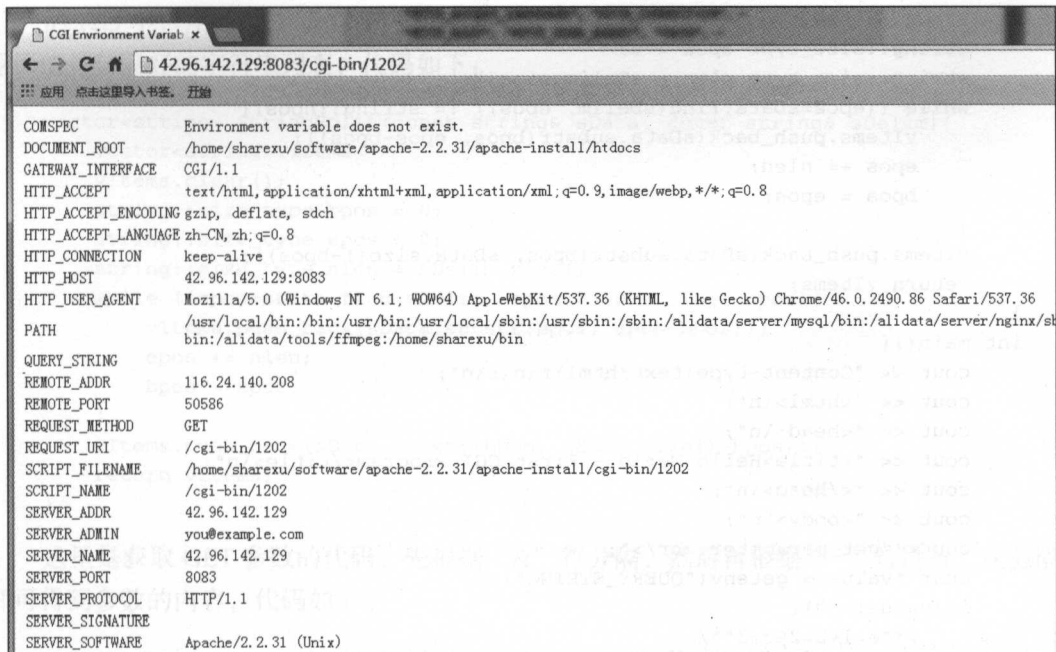


图 12-14 例 12.3 的 CGI 在浏览器执行结果图

如果再去把 CGI 程序手动执行一次，会发现结果还是和第一次手动执行该 CGI 程序的结果一样，这也验证了 CGI 是一个进程，且在处理完一个请求后就退出，在下一个请求到来时再创建一个新进程。



注意 REQUEST_METHOD 的值一般包括 POST 和 GET 两种。

4. GET 参数的获取

在 GET 方法下，CGI 程序无法直接从服务器的标准输入中获取数据，因为服务器把它从标准输入接收到的数据编码到环境变量 QUERY_STRING (或 PATH_INFO)。采用 GET 方法时，只需将这些数据附加到 URL 的末尾，如 `http://42.96.142.129:8083/cgi-bin/1203?a=1&b=2&c=3`，这时候 QUERY_STRING 的值为 `a=1&b=2&c=3`。

【例 12.3】 获取 GET 方法中的参数。

```
#include<iostream>
#include<stdlib.h>
#include<vector>
#include<string>
using namespace std;
vector<string> StringSplit(const string& sData, const string& sDelim){
    vector<string>vItems;
    vItems.clear();
```



```

string::size_type bpos = 0;
string::size_type epos = 0;
string::size_type nlen = sDelim.size();
while ((epos=sData.find(sDelim, epos)) != string::npos){
    vItems.push_back(sData.substr(bpos, epos-bpos));
    epos += nlen;
    bpos = epos;
}
vItems.push_back(sData.substr(bpos, sData.size()-bpos));
return vItems;
}

int main(){
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>Hello World - First CGI Program</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    cout<<"get parameter:<br/>";
    char *value = getenv("QUERY_STRING");
    if(value!= 0){
        /*a=1&b=2&c=3*/
        vector<string>paras=StringSplit((const string)value,"&");
        vector<string>::iterator iter=paras.begin();
        for(;iter!=paras.end();iter++){
            vector<string>singlepara=StringSplit(*iter,"=");
            cout<<singlepara[0]<<" "<<singlepara[1]<<"<br/>";
        }
        cout << "</body>\n";
        cout << "</html>\n";
        return 0;
    }
}

```

编译并复制到 cgi-bin 目录下，效果如图 12-15 所示。

```

[sharexu@linux 1203]$ g++ -o 1203 1203.cpp
[sharexu@linux 1203]$ cp 1203 /home/sharexu/software/apache-2.2.31/apache-install/cgi-bin

```

图 12-15 编译例 12.3 的程序和复制到 cgi-bin 目录下终端效果图

在浏览器中输入“http://42.96.142.129:8083/cgi-bin/1203?a=1&b=2&c=3”，按 Enter 键，即可得到如图 12-16 所示的结果。

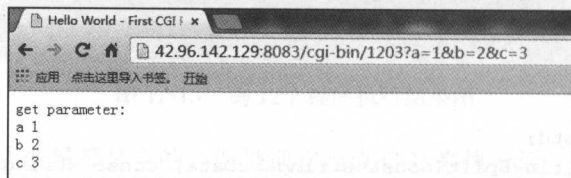


图 12-16 在浏览器中执行例 12.3 CGI 结果图

例 12.3 中，用了一个函数专门来分隔字符串的。sData 是源字符串，sDelim 是分隔符，返回值是分隔好的字符串列表，代码如下：

```
vector<string> StringSplit(const string& sData, const string& sDelim){
    vector<string>vItems;
    vItems.clear();
    string::size_type bpos = 0;
    string::size_type epos = 0;
    string::size_type nlen = sDelim.size();
    while ((epos=sData.find(sDelim, epos)) != string::npos){
        vItems.push_back(sData.substr(bpos, epos-bpos));
        epos += nlen;
        bpos = epos;
    }
    vItems.push_back(sData.substr(bpos, sData.size()-bpos));
    return vItems;
}
```

这里是获取 GET 参数的代码，先根据 “&” 符分隔，然后再根据 “=” 符进行二次分隔，即可得到参数的内容，代码如下：

```
vector<string>paras=StringSplit((const string)value,"&");
vector<string>::iterator iter=paras.begin();
for(;iter!=paras.end();iter++){
    vector<string>singlepara=StringSplit(*iter,"=");
    cout<<singlepara[0]<<" "<<singlepara[1]<<"<br/>";
}
```

5. POST 参数的获取

在 POST 方法下，CGI 程序可以直接从服务器的标准输入中获取数据，不过要先从 CONTENT_LENGTH 这个环境变量中得到 POST 参数的长度，然后再读取相应长度的内容。

【例 12.4】获取 POST 参数的内容。

post.html 的代码如下：

```
<html>
<head>
<title>CGI Post</title>
</head>
<body>
<tr><td>
<div style="font-weight:bold; font-size:15px">Method: POST</div>
<div>lease input two number:</div>
<form method="post" action="./cgi-bin/post">
<input type="txt" size="3" name="m">*
<input type="txt" size="3" name="n">=
<input type="submit" value="result">
</form>
</td></tr>
```

```

</table>
</body>
</html>

```

post.cpp 的代码如下：

```

#include<iostream>
#include<stdlib.h>
#include<stdio.h>
using namespace std;
int main(){
    cout << "Content-type:text/html\r\n\r\n";
    cout << "<html>\n";
    cout << "<head>\n";
    cout << "<title>Testing Post</title>\n";
    cout << "</head>\n";
    cout << "<body>\n";
    char *lenstr =getenv("CONTENT_LENGTH");
    if(lenstr==NULL){
        cout<<"Error, CONTENT_LENGTH should be entered!"<<"<br/>";
    }else{
        int len=atoi(lenstr);
        char poststr[20];
        fgets(poststr,len+1,stdin);
        cout<<"poststr:"<<poststr<<"<br/>";
        char m[10],n[10];
        if(sscanf(poststr,"m=%[^&]&n=%s",m,n)!=2){
            cout<<"Error: Parameters are not right!<br/>";
        }
        else{
            cout<<"m * n = "<<atoi(m)*atoi(n)<<"<br/>";
        }
    }
    cout << "</body>\n";
    cout << "</html>\n";
    return 0;
}

```

将 post.cpp 编译后放到 cgi-bin 目录下，把 post.html 复制到 htdocs 目录下，效果如图 12-17 所示。



```

[sharexu@linux 1204]$ vim post.cpp
[sharexu@linux 1204]$ g++ -o post post.cpp
[sharexu@linux 1204]$ cp post ../software/apache-2.2.31/apache-install/cgi-bin/
[sharexu@linux 1204]$ cp post.html ../../software/apache-2.2.31/apache-install/htdocs/
[sharexu@linux 1204]$

```

图 12-17 编译例 12.4 程序和复制到 cgi-bin 目录的终端效果图

在浏览器中输入“http://42.96.142.129:8083/post.html”，并按 Enter 键，得到如图 12-18 所示的内容。

分别在两个输入框中填入“3”和“4”，单击 result 按钮，会输出如图 12-19 所示的结果。

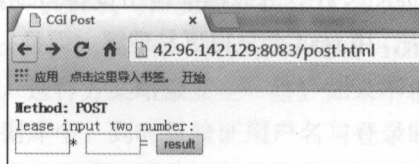


图 12-18 在浏览器中显式 post.html 的内容

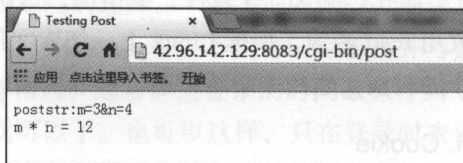


图 12-19 执行 post 这个 CGI 的内容图

例 12.4 中，先用 `getenv` 获得环境变量 `CONTENT_LENGTH` 的值，也就是有效信息的长度，代码如下：

```
char *lenstr =getenv("CONTENT_LENGTH");
```

然后用 `fgets` 函数从标准输入 `stdin` 中获得 `len+1` 长度的内容，代码如下：

```
fgets(poststr,len+1,stdin);
```

再利用 `sscanf` 函数获取 `m`、`n` 变量的值，代码如下：

```
sscanf(poststr,"m=%[^&]&n=%s",m,n)
```

对比例 12.3 和例 12.4，可以看到获得 GET 方法的参数和 POST 方法的参数的差异。

其实，例 12.4 的程序不够健壮，比如在读取 POST 参数内容，获取特定长度的字符串内容时，应当考虑读取中断的情况，原始代码如下：

```
int len=atoi(lenstr);
char poststr[20];
fgets(poststr,len+1,stdin);
```

这几行程序，可以改成下面这样：

```
string sPostData;
int len=atoll(lenstr);
if(len==0)return 0;
if(len>0){
    sPostData.resize(len);
    int bytes=0;
    while(bytes<len){
        int n=read(0,(char*)sPostData.data()+bytes, len-bytes);
        if(n<0){
            if(errno==EINTR) continue;
            return -1;
        }if(n==0){
            return -1;
        }else{
            bytes += n;
        }
    }
}
```

首先，参数的内容可能比较长，用 `atoll` 转换为长整型肯定比 `atoi` 转换为整型要靠谱些；其次，《UNIX 环境高级编程》中指出，每次调用 `fgets` 函数会造成标准输出设备自动刷新，这里改用 `read` 函数，根据 Linux 上一切皆文件的说法，用 `read` 会显得更易理解。而且，进行读操作的时候，遇到中断后，还可以继续读。

6. Cookie

在程序中，会话跟踪是很重要的事情。理论上，一个用户的所有请求操作都应该属于同一个会话，而另一个用户的所有请求操作则应该属于另一个会话，两者不能混淆。例如，用户 A 在超市购买的任何商品都应该放在 A 的购物车内，不论是用户 A 什么时间购买的，这都是属于同一个会话的，而不能放入用户 B 或用户 C 的购物车内，这不属于同一个会话。

而 Web 应用程序是使用 HTTP 协议传输数据的。HTTP 协议是无状态的协议。一旦数据交换完毕，客户端与服务器端的连接就会关闭，再次交换数据需要建立新的连接，这就意味着服务器无法从连接上跟踪会话。即用户 A 购买了一件商品并放入购物车内，当再次购买商品时服务器已经无法判断该购买行为是属于用户 A 的会话还是用户 B 的会话了。要跟踪该会话，必须引入一种机制。Cookie 就是这样的一种机制，它弥补了 HTTP 协议无状态的不足。

Cookie 实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用 `response` 向客户端浏览器颁发一个 Cookie。客户端浏览器会把 Cookie 保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该 Cookie 一同提交给服务器，服务器检查该 Cookie，以此来辨认用户状态。服务器还可以根据需要修改 Cookie 的内容。

很多网站都会使用 Cookie，例如 Google 会向客户端颁发 Cookie，Baidu 也会向客户端颁发 Cookie。那浏览器访问 Google 会不会也携带上 Baidu 颁发的 Cookie 呢？或者 Google 能不能修改 Baidu 颁发的 Cookie 呢？答案是否定的。Cookie 具有不可跨域名性。根据 Cookie 规范，浏览器访问 Google 只会携带 Google 的 Cookie，而不会携带 Baidu 的 Cookie；Google 也只能操作 Google 的 Cookie，而不能操作 Baidu 的 Cookie。Cookie 在客户端是由浏览器来管理的，从而保证用户的隐私安全。浏览器会判断一个网站是否能操作另一个网站 Cookie 的依据是域名，例如，Google 与 Baidu 的域名不一样，因此 Google 不能操作 Baidu 的 Cookie。需要注意的是，虽然网站 `images.google.com` 与网站 `www.google.com` 同属于 Google，但是域名不一样，两者同样不能互相操作彼此的 Cookie。

从表 12-1 可以看到，Cookie 也在环境变量中，可以用这样的程序获得 Cookie：

```
const char* cookie=getenv("HTTP_COOKIE");
```

如果用户是在自己家的计算机上上网，登录时就可以记住个人的登录信息，下次访问时不需要再次登录，直接访问即可。实现方法是把登录信息如账号、密码等保存在 Cookie 中，并控制 Cookie 的有效期，下次访问时再验证 Cookie 中的登录信息即可。保存登录信息

有多种方案。最直接的是把用户名与密码都保持到 Cookie 中，下次访问时检查 Cookie 中的用户名与密码，并与数据库比较。这是一种比较危险的选择，一般不把密码等重要信息保存到 Cookie 中。还有一种方案是把密码加密后保存到 Cookie 中，下次访问时解密并与数据库比较。这种方案略微安全一些。如果不希望保存密码，还可以把登录的时间戳保存到 Cookie 与数据库中，到时只验证用户名与登录时间戳就可以了。也可以这样，只在登录时查询一次数据库，以后访问验证登录信息时不再查询数据库。实现方式是把账号按照一定的规则加密后，连同账号一块保存到 Cookie 中。下次访问时只需要判断账号的加密规则是否正确即可。最后一种方案，也是现在各大网站的常用做法。

7. 获取用户的 IP 地址

有时候定位问题时，需要知道用户的 IP 地址，获取用户 IP 地址有两个环境变量：HTTP_VIA 和 REMOTE_ADDR。当用户使用代理服务器访问时，HTTP_QVIA 环境变量的值不为空，用户的 IP 也就是代理服务器的 IP 了。用户没有用代理服务器访问时，则 IP 地址就在 REMOTE_ADDR 环境变量中了。下面这个函数可以获得用户的 IP 地址。

```
std::string GetClientIP(){
    const char* p = getenv("HTTP_QVIA");           // proxy
    if(p&&strlen(p)>=8){
        char buf[32];
        snprintf(buf, sizeof(buf), "%d.%d.%d.%d",
            (uint8_t)hexToChar(p[0],p[1]),
            (uint8_t)hexToChar(p[2],p[3]),
            (uint8_t)hexToChar(p[4],p[5]),
            (uint8_t)hexToChar(p[6],p[7]));
        return std::string(buf);
    }
    else{
        p = getenv("REMOTE_ADDR");                 // no proxy
        if(p) return p;
    }
    return "0.0.0.0";
}
```

12.5 FastCGI

CGI 工作原理：每当客户请求 CGI 的时候，Web 服务器就请求操作系统生成一个新的 CGI 进程，该进程处理完请求后退出，下一个请求来时再创建新进程。当然，这样在访问量很少没有并发的情况可行，可是当访问量增大且并发存在时，这种方式就不适合了，于是就有了 FastCGI。

FastCGI 像是一个常驻 (long-live) 型的 CGI，它可以一直执行着，只要激活后，不会每次都要花费时间去 fork 一次 (这是 CGI 最为人诟病的 fork-and-execute 模式)。

一般情况下，FastCGI 的整个工作流程如下所述。

- (1) Web Server 启动时载入 FastCGI 进程管理器 (IIS ISAPI 或 Apache Module)。
- (2) FastCGI 进程管理器自身初始化，启动多个 CGI 进程并等待来自 Web 服务器的连接。
- (3) 当客户端请求到达 Web Server 时，FastCGI 进程管理器选择并连接到一个 FastCGI 进程。Web 服务器将 CGI 环境变量和标准输入发送到 FastCGI 进程。
- (4) FastCGI 子进程完成处理后将标准输出和错误信息从同一连接返回 Web Server。当 FastCGI 子进程关闭连接时，请求便被告知处理完成。FastCGI 进程接着等待并处理来自 FastCGI 进程管理器 (运行在 Web 服务器中) 的下一个连接。

对 FastCGI 有兴趣的读者可以去网上搜索更详细的资料。

12.6 本章小结

本章主要介绍了 HTTP 协议及其应用 CGI 的环境变量、GET 方法的使用、POST 方法的使用、Cookie 的使用和用户 IP 地址的获取等。接下来的第 13 章，将会介绍下后台开发中常用的类库。



第 13 章 Chapter 13

常用类库

我们在编写程序的过程中经常会遇到代码重复的情况，可以在重复时选择复制粘贴，或者构建一个函数来实现复用。但是这样并不是最好的解决办法，因为出现这种情况是由于一开始就不完全清楚这个过程是怎么样的，对功能也没有进行深入分析。在开发之前，只是了解了大概需求，然后就开始编程，在开发过程当中东拼西凑，虽然仍然能够把功能完成，但是对整个程序并不能了然于胸，出现问题就一句一句地查找问题，显然这是很浪费时间的。

如何解决这个问题，答案就是类化。在编写代码之前，先将整个系统的功能做个整体的认识，然后对过程功能进行抽象，在抽象的过程当中就需要对各个功能的各个细节进行考虑，将功能相关的函数最简化。由于在编写代码之前就已经对细节进行了考查，所以在编写过程中就不会出现重复代码的情况，当然也有一个粒度问题，比如在某个参数有两个输入，产生 4 种可能，而每一种可能当中有些细节是相同的，那么到底应不应该将这些细节提取出来进行复用，这就是粒度问题，粒度问题应该结合实际和自身的情况来做决定。接着就可以将这个功能类化，功能类化的优点不仅仅可以实现类的复用（因为在设计之前每个细节都已经考虑到），更关键的地方在于使用这个类的开发人员不需要知道内部具体的实现是怎么样的，这就是笔者认为要实现类应该达到的目标。

构建一个类库，首先需要提取功能、抽象功能，然后设计私有数据、公共数据以及方法，完成了这些工作之后，一个类库就完成了。之后把头文件、实现拿出来，放在程序中，完成复用。当然底层系统不同具体的函数是需要重新实现的，但上层代码完全不需要更改。

要使用第三方源码库，第一步少不了的就是编译，将源码文件编译成方便使用的动态链接库、静态链接库或者静态导入库。

使用第三方源码最简单的方法是直接将文件加入工程，但这样不利于软件、源码产品的管理，对于一般软件开发来说，不建议使用。

C++ 已经有这么多年的历史了，很多类库都已经很完善了，本章将介绍几个常用的类库。

13.1 JSON

前面第 12 章讲到，web 页面通常使用 CGI 来与后端交互。假设有这样的场景，页面需要展示当前用户的数据，比如年龄 age，这时候 CGI 应该怎么样返回数据？实际上，页面拿到 CGI 返回的数据后，需要逐个解析每个字段的值，再将其展示到页面上。当用户的数据有多个，比如职业 (occupation) 和性别 (sex)，那就得是 age=30&occupation=engineer&sex=female，如果有两个用户，则更加复杂：age=30&occupation=engineer&sex=female&age=31&occupation=engineer&sex=male，可以看到数据解析麻烦，也不好扩展。这时候 JSON 就可以派上用场了，一个用户的数据可以表示为：{"age":30}，乍一看，好像没什么优势，甚至比 age=30 更占空间，但是当多个 key-value 串在一起时，JSON 就能体现它的价值了。比如一个用户的多维数据可以这样表示：

```
{
  "age":30,
  "occupation":"engineer",
  "sex":"female"
}
```

两个用户的多维数据可以这样表示：

```
{
  "users":[
    {
      "age":30,
      "occupation":"engineer",
      "sex":"female"
    },
    {
      "age":31,
      "occupation":"engineer",
      "sex":"male"
    }
  ]
}
```

是不是很清楚明了，而且，JSON 还有专门的 API，帮助开发者解析出各个字段的值。下面具体来看下 JSON 究竟是什么。

1. JSON 是什么

JSON (JavaScript Object Notation, JavaScript 对象表示法) 是一种轻量级的数据交换格式, 易于人阅读和编写, 同时也易于机器解析和生成。JSON 是存储和交换文本信息的语法, 采用完全独立于语言的文本格式, 但是也使用了类似于 C 语言家族的习惯 (包括 C、C++、C#、Java、JavaScript、Perl、Python 等)。这些特性使 JSON 成为理想的数据交换语言。

JSON 构建于两种结构, 如下所述。

(1) key-value 对的集合。不同的语言中, 它被分别理解为对象 (object)、纪录 (record)、结构 (struct)、字典 (dictionary)、哈希表 (hash table)、有键列表 (keyed list) 或者关联数组 (associative array)。

(2) 值的有序列表。在大多数语言中, 它被理解为数组 (array)、矢量 (vector)、列表 (list) 或者是序列 (sequence)。

JSON 对象是一个无序的 key-value 集合。一个 JSON 对象以 “{” (左括号) 开始, “}” (右括号) 结束。每个 key 后跟一个 “:” (冒号); key-value 之间使用 “,” (逗号) 分隔。

数组是值 (value) 的有序集合。一个数组以 “[” (左中括号) 开始, “]” (右中括号) 结束。值之间使用 “,” (逗号) 分隔。

值 (value) 可以是双引号括起来的字符串 (string)、数值 (number)、true、false、null、对象 (object) 或者数组 (array), 并且这些结构可以嵌套。

字符串 (string) 是由 0 到多个 unicode 字符组成的序列, 封装在双引号 (“”) 中, 可以使用反斜杠 (\) 来进行转义。一个字符可以表示为一个单一字符的字符串。

数字 (number) 类似 C 或者 Java 里面的数, 没有用到的 8 进制和 16 进制数除外。

2. JsonCpp 的使用

JsonCpp 是 C++ 中比较稳定的处理 JSON 的库, 要使用这类第三方源码库, 第一步少不了的就是编译, 将源码文件编译成方便使用的动态链接库、静态链接库或者静态导入库。JsonCpp 进行 JSON 解析的源码文件分布在 include/json、src/lib_json 下。其实 JsonCpp 源码并不多, 为了方便产品管理, 此处没必要将其编译为动态链接库或者静态导入库, 所以笔者选择使用静态链接库。

这里编译 JsonCpp 的静态库需要用到一个工具 scons, 先用已 root 的用户权限执行 yum install scons 命令来安装 scons, 等到提示 “Complete!” 就是安装成功了。解压 JsonCpp 的压缩包, 进入解压目录后, 执行 “scons platform=linux-gcc” 命令, 它会自行编译, 编译出来的库文件在其 libs/linux-gcc-4.4.6 目录下, 有 libjson_linux-gcc-4.4.6_libmt.so 和 libjson_linux-gcc-4.4.6_libmt.a; 头文件在解压目录下的 include 中。再把 libjson_linux-gcc-4.4.6_libmt.a 和头文件复制到所要编码的目录中, 这时候就可以开始编写测试程序了。

【例 13.1】 JsonCpp 的使用范例。

test.cpp 的代码是：

```

#include <iostream>
#include <string>
#include "json/json.h"
using namespace std;

int main(){
    Json::Value json_temp;
    json_temp["name"] = Json::Value("sharexu");
    json_temp["age"] = Json::Value(18);
    Json::Value root;
    root["key_string"] = Json::Value("value_string");
    root["key_number"] = Json::Value(12345);
    root["key_boolean"] = Json::Value(false);
    root["key_double"] = Json::Value(12.345);
    root["key_object"] = json_temp;
    root["key_array"].append("array_string");
    root["key_array"].append(1234);

    Json::FastWriter fast_writer;
    std::cout << fast_writer.write(root);

    Json::StyledWriter styled_writer;
    std::cout << styled_writer.write(root);

    string str_test = "{\"id\":1,\"name\":\"pacozhong\"}";
    Json::Reader reader;
    Json::Value value;
    if (!reader.parse(str_test, value))
        return 0;
    string value_name=value["name"].asString();
    cout <<value_name<<endl;
    cout <<value["name"];
    if(!value["id"].isInt()){
        cout<<"id is not int"<<endl;
    }else{
        int value_id=value["id"].asInt();
        cout<<value_id<<endl;
    }
    return 0;
}

```

makefile 的代码是：

```

test:test.o
g++ -o test test.o -I./include -L./lib -ljson_linux-gcc-4.4.6_libmt
test.o:test.cpp
g++ -c test.cpp -I./include -L./lib -ljson_linux-gcc-4.4.6_libmt

```

执行 make 命令，即可编译成功。执行 ./test 命令，程序的执行结果如图 13-1 所示。

```
[sharexu@linux 1301]$ ./test
{"key_array":["array_string",1234],"key_boolean":false,"key_double":12.3450,"key_number":12345,"key_object":{
  "age":18,"name":"sharexu"},"key_string":"value_string"}
{
  "key_array": [ "array_string", 1234 ],
  "key_boolean" : false,
  "key_double" : 12.3450,
  "key_number" : 12345,
  "key_object" : {
    "age" : 18,
    "name" : "sharexu"
  },
  "key_string" : "value_string"
}
pacozhong
pacozhong
[sharexu@linux 1301]$
```

图 13-1 例 13.1 程序的执行结果

注意，当前的目录树应该如图 13-2 所示。

JsonCpp 主要包含 3 种类型的类：Value、Reader、Writer。JsonCpp 中所有对象、类名都在命名空间 JSON 中，使用时包含 json.h 即可。Json::Value 是 JsonCpp 中最基本、最重要的类，用于表示各种类型的对象，JsonCpp 支持的对象类型可见 Json::ValueType 枚举值。

例 13.1 中，声明了一个 Json::Value 对象 json_temp，并赋值了 2 个 key-value 对：name:sharexu 及 age:18，代码如下：

```
Json::Value json_temp;
json_temp["name"] = Json::Value("sharexu");
json_temp["age"] = Json::Value(18);
```

```
[sharexu@linux 1301]$ tree
.
├── include
│   └── json
│       ├── autolink.h
│       ├── config.h
│       ├── features.h
│       ├── forwards.h
│       ├── json.h
│       ├── reader.h
│       ├── value.h
│       └── writer.h
├── lib
│   └── libjson_linux-gcc-4.4.6_libmt.a
├── makefile
├── test
│   ├── test.cpp
│   └── test.o
└── 3 directories, 13 files
[sharexu@linux 1301]$
```

图 13-2 引用 JsonCpp 时的目录树

再声明一个 Json::Value 对象 root，并赋值了字符串、数值、布尔值、浮点数、Json::Value 对象、数组 6 种类型，如下所示。

```
Json::Value root;
root["key_string"] = Json::Value("value_string");
root["key_number"] = Json::Value(12345);
root["key_boolean"] = Json::Value(false);
root["key_double"] = Json::Value(12.345);
root["key_object"] = json_temp;
root["key_array"].append("array_string");
```

添加一个数组的值，用 append 函数即可，代码如下：

```
root["key_array"].append(1234);
```

Jsoncpp 的 Json::Writer 类是一个纯虚类，并不能直接使用。在此使用 Json::Writer 的子类：Json::FastWriter、Json::StyledWriter、Json::StyledStreamWriter。其中 Json::FastWriter 来处理 JSON 是最快的。

```
Json::FastWriter fast_writer;
std::cout << fast_writer.write(root);
```

以上两行代码的执行结果是：

```
{"key_array":["array_string",1234],"key_boolean":false,"key_double":12.3450,"key_number":12345,"key_object":{"age":18,"name":"sharexu"},"key_string":"value_string"}
```

用 `Json::StyledWriter` 得到的是格式化后的 JSON，下面来看看 `Json::StyledWriter` 是怎样格式化的，代码如下：

```
Json::StyledWriter styled_writer;
std::cout << styled_writer.write(root);
```

以上两行代码的执行结果是：

```
{
  "key_array" : [ "array_string", 1234 ],
  "key_boolean" : false,
  "key_double" : 12.3450,
  "key_number" : 12345,
  "key_object" : {
    "age" : 18,
    "name" : "sharexu"
  },
  "key_string" : "value_string"
}
```

`Json::Reader` 是用于读取的，确切地说，是用于将字符串转换为 `Json::Value` 对象的。下面就是把 `str_test` 字符串转换成 JSON 对象 `value` 的。注意，字符串中的双引号要用反斜杠 (\) 进行转义，代码如下：

```
string str_test = "\\\"id\\\":1,\\\"name\\\":\\\"pacozhong\\\"\"";
Json::Reader reader;
Json::Value value;
if (!reader.parse(str_test, value))
    return 0;
```

获取 JSON 对象中的元素的值有 2 种方法，比如 `value["name"].asString()` 和 `value["name"]`，代码如下：

```
string value_name=value["name"].asString();
cout <<value_name<<endl;
cout <<value["name"];
```

但是，用 `as*` 的方法取值时，注意判断类型是否准确，比如要 `value["id"].asInt()` 取得 `id` 的整型值，就得先判断 `value["id"]` 的值是否是整型的，否则程序可能产生异常，代码如下：

```
if(!value["id"].isInt()){
    cout<<"id is not int"<<endl;
}else{
    int value_id=value["id"].asInt();
    cout<<value_id<<endl;
```


JSON 用于数据传输时，基本上是先转换成字符串形式后再进行传输，使用方拿到字符串后再解析出 JSON，得到各个字段的内容。

3. JSON 用途

这里要提到两个概念，序列化和反序列化。序列化是将对象状态转换为可保持或传输的格式的过程。与序列化相对的是反序列化，它将流转换为对象。这两个过程结合起来，可以轻松地数据进行存储和传输。

由于很多页面都是用 Java Script 写的，而使用 Java Script 解析 JSON 又非常方便，所以很多 CGI 都是用 JSON 与页面进行通信的。

JSON 可以以字符串的形式存储，要使用时再进行序列化。

13.2 Protobuf

与 JSON 相比，Protobuf 的序列化和反序列化的速度更快，而且传输的数据会先压缩，使得传输的效率更高些。

Protobuf，全称 Protocol Buffer，是 Google 公司内部的混合语言数据标准，是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。它很适合做数据存储或 RPC 数据交换格式。Protobuf 是可用于通信协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。目前提供了 C++、Java、Python 三种语言的 API。

1. 一个简单的例子

要使用 Protobuf，需要先安装得到 protoc 文件，这是用来生成头文件和 .cc 文件的工具。在网站 <http://code.google.com/p/protobuf/downloads/list> 上可以下载 Protobuf 的源代码。然后解压、编译安装便可以使用它了。

安装中的命令步骤如下所示。

```
tar -xzf protobuf-2.5.0.tar.gz
cd protobuf-2.5.0
./configure --prefix= /home/sharexu/software/protobuf/protobuf-install
make
make check
make install
```

可以看到在 /home/sharexu/software/protobuf/protobuf-install 目录下有 bin、include 和 lib 目录。可以把 include 目录下的文件都按照该目录结构和 lib/libprotobuf.a 复制到所需要的目录中去，这样就可以开始写测试程序了。

【例 13.2】Protobuf 的使用范例。

Mymessage.proto 的代码是：

```

package Im;
message Content
{
    required int32    id = 1;           // ID
    required string   str = 2;         // str
    optional int32    opt = 3;         // optional field
}

```

Writer.cpp 的代码是:

```

#include<iostream>
#include<fstream>
#include "Mymessage.pb.h"
using namespace std;
int main(){
    Im::Content msg1;
    msg1.set_id(101);
    msg1.set_str("sharexu");
    fstream output("./log", ios::out | ios::trunc | ios::binary);
    if (!msg1.SerializeToOstream(&output)) {
        cerr << "Failed to write msg." << endl;
        return -1;
    }
    return 0;
}

```

Reader.cpp 的代码是:

```

#include<iostream>
#include<fstream>
#include "Mymessage.pb.h"
using namespace std;
void ListMsg(const Im::Content & msg){
    cout << msg.id() << endl;
    cout << msg.str() << endl;
}
int main(int argc, char* argv[]){
    Im::Content msg1;
    fstream input("./log", ios::in | ios::binary);
    if (!msg1.ParseFromIstream(&input)) {
        cerr << "Failed to parse address book." << endl;
        return -1;
    }
    ListMsg(msg1);
    return 0;
}

```

makefile 的代码是:

```

INC=/home/sharexu/charpter13/1302/include
LIB=/home/sharexu/charpter13/1302/lib

```

```

lib=protobuf

all:Writer Reader

Writer.o:Writer.cpp
    g++ -g -c Writer.cpp -I$(INC) -L$(LIB) -l$(lib)
Reader.o:Reader.cpp
    g++ -g -c Reader.cpp -I$(INC) -L$(LIB) -l$(lib)

Writer:Writer.o Mymessage.pb.o
    g++ -g -o Writer Writer.o Mymessage.pb.o -I$(INC) -L$(LIB) -l$(lib)
Reader:Reader.o Mymessage.pb.o
    g++ -g -o Reader Reader.o Mymessage.pb.o -I$(INC) -L$(LIB) -l$(lib)
Mymessage.pb.o:Mymessage.pb.cc
    g++ -g -c Mymessage.pb.cc -I$(INC) -L$(LIB) -l$(lib)

clean:Writer Reader Writer.o Reader.o Mymessage.pb.o
    rm Writer Reader Writer.o Reader.o Mymessage.pb.o

```

执行 `/home/sharexu/software/protobuf/protobuf-install/bin/protoc -I=./ --cpp_out=./ Mymessage.proto` 命令后，会生成 `Mymessage.pb.h` 和 `Mymessage.pb.cc` 文件。再执行 `make` 命令，生成 `Writer` 和 `Reader` 文件。执行 `./Writer` 命令后，再执行 `./Reader` 命令，终端上输出：

```

101
sharexu

```

例 13.2 的程序由两部分组成，第一部分被称为 `Writer`，第二部分叫作 `Reader`。`Writer` 负责将一些结构化的数据写入一个磁盘文件，`Reader` 则负责从该磁盘文件中读取结构化数据并打印到屏幕上。准备用于演示的结构化数据是 `Content`，它包含两个基本数据：① `id`，为一个整数类型的数据；② `str`，是一个字符串。结构化数据是定义在 `Mymessage.proto` 文件中的。`proto` 文件非常类似 C++ 中的数据定义。`Mymessage.proto` 中，定义了一个消息 `Content`，该消息有 3 个成员：① 类型为 `int32` 的 `id`；② 类型为 `string` 的成员 `str`；③ `opt` 是一个可选的成员，即消息中可以不包含该成员。

```

package Im;
message Content
{
    required int32    id = 1;           // ID
    required string   str = 2;          // str
    optional int32    opt = 3;          // optional field
}

```

写好 `proto` 文件之后就得用 `Protobuf` 编译器将该文件编译成目标语言了。执行 `/home/sharexu/software/protobuf/protobuf-install/bin/protoc -I=./ --cpp_out=./ Mymessage.proto` 命令生成 `Mymessage.pb.h` 和 `Mymessage.pb.cc` 文件。`Mymessage.pb.h` 定义了 C++ 类的头文件 `Mymessage.pb.cc` 文件定义了 C++ 类的实现文件。在生成的头文件中，定义了一个 C++ 类 `Content`，后面

的 Writer 和 Reader 将使用这个类来对消息进行操作。诸如对消息的成员进行赋值，将消息序列化等都有相应的方法。

Writer 将把一个结构化数据写入磁盘，以便其他用户来读取。假如不使用 Protobuf，其实也有许多其他的选择：比如可以将数据转换为字符串，然后将字符串写入磁盘。转换为字符串的方法可以使用 `sprintf()`，这非常简单。数字 123 可以变成字符串 123。这样做似乎没有什么不妥，但是仔细考虑一下就会发现，这样的做法对 Reader 的要求比较高，Reader 的作者必须了解 Writer 的细节。比如“123”可以是单个数字 123，但也可以是 3 个数字 1、2 和 3，等等。这么说来，还必须让 Writer 定义一种分隔符一样的字符，以便 Reader 可以正确读取。最后发现一个简单的 Content 也需要写许多处理消息格式的代码。如果使用 Protobuf，那么这些细节就可以不需要应用程序来考虑了。

Protobuf 使 Writer 的工作变得很简单，需要处理的结构化数据由 .proto 文件描述，经过上一节中的编译过程后，该数据化结构对应了一个 C++ 的类，并定义在 `Mymessage.pb.h` 中。对于本例，类名为 `Im::Content`。Writer 需要 include 该头文件，然后便可以使用这个类了。

现在在 Writer 代码中，将要存入磁盘的结构化数据由一个 `Im::Content` 类的对象表示，它提供了一系列的 `get/set` 函数 (field) 用来修改和读取结构化数据中的数据成员。

当需要将该结构化数据保存到磁盘上时，类 `Im::Content` 已经提供相应的方法来把一个复杂的数据变成一个字节序列，可以将这个字节序列写入磁盘。对于想要读取这个数据的程序来说，也只需要使用类 `Im::Content` 的相应反序列化方法来将这个字节序列重新转换成结构化数据。这同开始时那个“123”的想法类似，不过 Protobuf 想的远远比之前那个粗糙的字符串转换要全面，因此，可以放心将这类事情交给 Protobuf。

下面是定义一个 `Im::Content` 对象，`set_id()` 用来设置 id 的值，代码如下：

```
Im::Content msg1;
msg1.set_id(101);
msg1.set_str("hello");
```

下面把 `msg1` 的内容序列化后存储当前目录的 `log` 文件中。其中 `SerializeToOstream` 就是将对象序列化，代码如下：

```
fstream output("./log", ios::out | ios::trunc | ios::binary);
if (!msg1.SerializeToOstream(&output)) {
    cerr << "Failed to write msg." << endl;
    return -1;
}
```

而对于 Reader，只需要从 `log` 文件中读取，反序列化后就能获得结构化的数据。Reader 声明类 `helloworld` 的对象 `msg1`，然后利用 `ParseFromIstream` 从一个 `fstream` 流中读取信息并反序列化，代码如下：

```

Im::Content msg1;
fstream input("./log", ios::in | ios::binary);
if (!msg1.ParseFromIstream(&input)) {
    cerr << "Failed to parse address book." << endl;
    return -1;
}

```

此后，ListMsg 中采用 get 方法读取消息的内部信息，并进行打印输出操作，代码如下：

```

void ListMsg(const Im::Content & msg){
    cout << msg.id() << endl;
    cout << msg.str() << endl;
}

```

这个例子本身并无实际意义，但只要稍加修改就可以将它变成更加有用的程序。比如将磁盘替换为网络 socket，那么就可以实现基于网络的数据交换任务，而存储和交换正是 Protobuf 最有效的应用领域。

2. 高效率的 Protobuf

Protobuf 的高效率表现在以下两个方面。

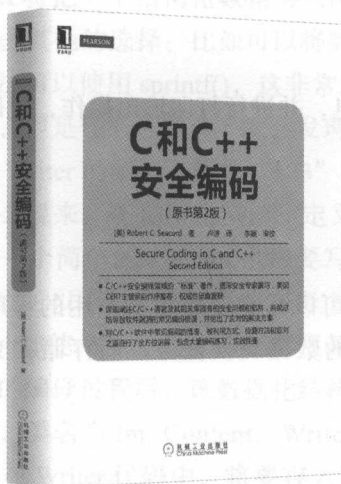
(1) Protobuf 序列化后的信息内容表示非常紧凑，减少了消息的体积，自然只需要更少的资源。比如网络上传输的字节数更少，需要的 IO 设备更少等，从而提高性能。

(2) Protobuf 封解包的速度更快。

13.3 本章小结

本章从数据传输和压缩的角度简单介绍了 JSON 和 Protobuf，包括它们解决了什么问题和使用方法等。比 JSON 的效率更高的还有腾讯的开源库 RapidJSON 等，有兴趣的读者可以自行了解。

推荐阅读



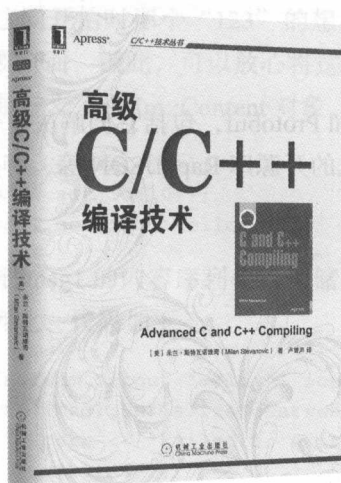
C和C++安全编码 (原书第2版)

作者: Robert C. Seacord ISBN: 978-7-111-44279-0 定价: 79.00元



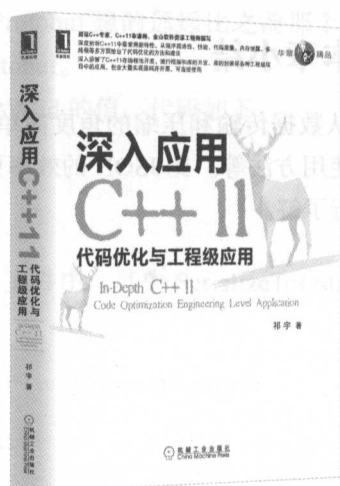
大规模C++程序设计

作者: John Lakos ISBN: 978-7-111-47425-8 定价: 129.00元



高级C/C++编译技术

作者: 米兰·斯特瓦诺维奇 ISBN: 978-7-111-49618-2 定价: 69.00元



深入应用C++11: 代码优化与工程级应用

作者: 祁宇 ISBN: 978-7-111-50069-8 定价: 79.00元

互联网网民日益剧增，各种应用层出不穷，各项技术更新不断。单是游戏行业，近几年就经历了从端游、页游到手游的巨大变迁，客户端更新迭代之快，始料未及。而后台开发中使用到的技术，却变化不是很大。让服务性能更高、处理能力更强、安全性更好，是后台开发工程师永恒的主题。

后台开发中用到的技术，深而广，需要读的“大部头”很多，光是Richard Stevens的APUE，UNP，TCP/IP详解就够读个半年以上。读者通过阅读本书，可以从实践出发，快速由浅入深地进入后台开发领域。在读完本书，有了实践的经验之后，再去阅读大师们的著作，会更有体会，更懂得如何欣赏。

读书的最高境界莫过于“把书读薄，把书读厚”。本书文字通俗易懂，让你更快地“读薄”，同时又涉及较多的核心知识点，顺着这些知识点，读着读着也发觉“读厚”了。

作者简介

徐晓鑫 腾讯资深软件研发工程师，先后在腾讯游戏之洛克王国、QQ会员、QQ秀等项目工作，精通后台开发各种技术，实战经验丰富。



后台开发是一个“历史悠久”的领域，同时也是一个沉淀深厚，高技术价值的领域。本书清晰、严谨、务实的风格显示出晓鑫对该领域知识的深刻理解。

——张子兴 Facebook对外支付项目主程，美国加州MenloPark

每一位从事后台开发的专业人士都需要一本《后台开发：核心技术与应用实践》。对每一位想要认真从事该领域工作的人来说，这是一本绝对必读的书籍。

——彭可竞 微软软件工程师，美国华盛顿州Redmond

本书是作者多年后台开发、架构和研究的精华。书中用通俗的文字、详尽的示例代码，结合实际工作中的案例，讲述了后台开发方方面面的知识，内容丰富。对于从事后台开发的人员，这是一本很好的由浅入深的学习书籍。

——周乐 阿里巴巴资深算法工程师，北京望京

使用C++语言进行后台开发有一定的门槛，本书可以很好地帮助你跨过这个“门槛”。

——畅晋 百度大数据高级测试工程师，北京上地



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机\程序设计

ISBN 978-7-111-54339-8



9 787111 543398 >

定价: 79.00元